UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

TÍTULO: Aceleración del Algoritmo Floyd-Warshall sobre Intel Xeon Phi KNL

AUTORES: Ulises Costi

DIRECTOR: Enzo Rucci, Franco Chichizola

CODIRECTOR:

ASESOR PROFESIONAL:

CARRERA: Licenciatura en Informática

Resumen

En la última década, los aceleradores han tomado mayor protagonismo en la comunidad de HPC. Recientemente, Intel introdujo la segunda generación de aceleradores Xeon Phi, con nombre en código Knights Landing (KNL), el cual trae importantes mejoras con respecto a su predecesor. Esta tesina se enfoca en evaluar el uso de la arquitectura KNL para acelerar el algoritmo Floyd-Warshall (FW). Partiendo de una versión paralela “clásica” del mismo, se muestra cómo aumenta el rendimiento con cada optimización aplicada hasta llegar a la solución óptimizada, con la cual se logró un pico de 1039 GFLOPS.

Palabras Clave

Xeon Phi, Knights Landing, Floyd-Warshal, AVX-512, MCDRAM, HPC, Hyper-Threading, grafo, caminos mínimos.

Conclusiones

El objetivo planteado en esta tesina se considera satisfecho al lograr una versión de FW optimizada para Xeon Phi KNL, acompañada del análisis e investigación que fueron necesarios para su desarrollo. Se investigó acerca de la arquitectura, el algoritmo, y sobre las diferentes técnicas de optimización. También fueron desarrolladas versiones variantes del programa, con el fin de evaluar su utilidad en distintos escenarios alternativos. Para beneficio de la comunidad, el código se encuentra disponible en un repositorio público.

Trabajos Realizados

Se presentó la arquitectura Xeon Phi KNL, se enunció el problema del camino mínimo en grafos, y al algoritmo FW como una de sus soluciones. Con el fin de lograr una versión de FW optimizada para KNL, se llevó a cabo un desarrollo incremental iterativo; partiendo del algoritmo base, se aplicaron diferentes técnicas de optimización comparando los resultados de rendimiento en cada caso. En una segunda etapa, se desarrollaron versiones variantes del programa, a modo de evaluar diferentes alternativas y parámetros.

Trabajos Futuros

Por un lado, se planea continuar refinando la optimización del programa desarrollado en este trabajo, ya que es probable que se le pueda extraer aún más rendimiento al KNL con este algoritmo.
Por otro lado, siendo las GPU los aceleradores dominantes en la actualidad, resulta interesante utilizar FW para comparar el rendimiento entre ambas arquitecturas.

Fecha de la presentación: Sep 2020

Aceleración del Algoritmo Floyd-Warshall sobre Intel Xeon Phi KNL

Ulises Costi

Facultad de Informática

Universidad Nacional de La Plata

Director: Dr. Enzo Rucci

Director: Esp. Franco Chichizola

Tesina presentada para obtener el grado de

Licenciado en Informática

Septiembre 2020

Resumen

Desde hace años, los aceleradores van tomando mayor protagonismo en la comunidad de HPC. Con la introducción de los Xeon Phi de segunda generación, con nombre en código Knights Landing (KNL), la comunidad dispone de un nuevo acelerador x86 que trae importantes mejoras con respecto a su predecesor. Entre ellas se destacan la ejecución fuera de orden, la duplicación de la cantidad de VPUs, y la integración de una memoria de alto ancho de banda.

Una de las áreas bien conocidas por demandar gran poder de cómputo es la teoría de grafos, siendo el algoritmo Floyd-Warshall (FW) un caso bien conocido de la misma. La popularidad y alta demanda computacional de FW lo vuelve un caso interesante de análisis en HPC. Es por este motivo que esta tesina se enfoca en evaluar el uso de arquitectura Xeon Phi KNL para acelerar el algoritmo FW. Partiendo de una versión paralela “clásica” de FW, se muestra cómo aumenta el rendimiento con cada optimización aplicada hasta llegar a la solución optimizada, con la cual se logró un pico de 1039 GFLOPS. Complementariamente, se analizaron diferentes variantes de la implementación, con el fin de evaluar su utilidad en distintos escenarios alternativos. Por último, el código se encuentra disponible para beneficio de la comunidad académica, científica y productiva.

Agradecimientos

A mi madre, por el esfuerzo y acompañamiento durante todos estos años.

A mis abuelos y mi novia, por la colaboración y el apoyo incondicional.

A Víctor, Guillermo, Gustavo y Carlos, por ser mi inspiración para elegir esta carrera.

A Enzo y Franco, por su labor excepcional como directores de esta tesina.

A la Facultad de Informática y el LIDI, por brindarme los medios y recursos para realizar este trabajo.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En primer lugar, se presenta la motivación de esta tesina (sección 1.1). Luego se enuncian los objetivos y metodología (sección 1.2), y las contribuciones y resultados obtenidos (sección 1.3). Por último, se describe la organización del documento (sección 1.5).

1.1. Motivación

El problema del consumo energético se presenta como uno de los mayores obstáculos para el diseño de sistemas de Exa-escala. Por lo tanto, la comunidad científica está en la búsqueda de diferentes maneras de mejorar la eficiencia energética de los sistemas de cómputo de altas prestaciones (HPC por sus siglas en inglés). Una de las estrategias más elegidas en los últimos años ha sido la de incorporar arquitecturas de aceleradores y coprocesadores a los sistemas de supercómputo. La clave en el uso de estos dispositivos se basa en que incrementan el poder computacional de un sistema al mismo tiempo que limitan su consumo de potencia, lo que les permite obtener mejores prestaciones desde la perspectiva energética [1] [2].

Los aceleradores más usados suelen ser las unidades de procesamiento gráfico (GPU por sus siglas en inglés) o los coprocesadores Xeon Phi de Intel. Recientemente, Intel ha presentado la segunda generación de esta clase de aceleradores, con nombre clave Knigths Landing (KNL). Entre sus principales características, se pueden mencionar la gran cantidad de núcleos con soporte para Hyper-Threading, la incorporación de las instrucciones vectoriales AVX-512 y la integración de una memoria de gran ancho de banda.

Desde sus inicios, los grafos y sus algoritmos se han vuelto muy importantes, ya que permiten modelar y resolver problemas de dominios muy diferentes como computación científica, minería de datos, procesamiento de imágenes, ruteo de redes, entre otros. No obstante, estos algoritmos por lo general tienen un costo computacional relativamente elevado, por lo que resulta apropiado ejecutarlos sobre un hardware acorde a las necesidades. Con el objetivo de reducir el tiempo de ejecución al máximo posible, no resulta suficiente simplemente portar la aplicación al nuevo hardware sino que se la debe adaptar de acuerdo a sus características específicas.

Entre los algoritmos de grafos descritos en el párrafo anterior, se encuentra Floyd-Warshall (FW) [3] [4], el cual permite calcular (y hallar) los caminos mínimos entre todos los vértices de un grafo pesado. A este algoritmo en toda su historia se lo ha empleado en ámbitos diversos como el tráfico automovilístico [5], las redes de computadoras [6], bioinformática [7], entre otros. Sin embargo, FW es computacionalmente costoso ($O(n^3)$) y a medida que el tamaño del

problema escala, el empleo de recursos de cómputo paralelo se vuelve necesario para poder satisfacer sus requerimientos.

De manera de poder computar los caminos mínimos de un grafo con tiempos de respuesta aceptables, resulta necesario desarrollar nuevas soluciones computacionales que sean capaces de aprovechar las arquitecturas HPC actuales.

1.2. Objetivos y metodología

El objetivo de esta tesina consiste en evaluar el uso de la arquitectura Xeon Phi KNL para acelerar el algoritmo FW. Para ello se realizarán las siguientes actividades:

- Se estudiará el problema de calcular los caminos mínimos en un grafo y sus requisitos computacionales, en particular siguiendo el algoritmo FW.
- Se examinará la arquitectura de los aceleradores Xeon Phi KNL, los modelos de programación y las técnicas de optimización aplicables.
- Se relevará la bibliografía existente en la temática a partir de la búsqueda en bases de datos especializadas.
- Se diseñarán y desarrollarán diferentes soluciones paralelas al problema estudiado que puedan ejecutarse en aceleradores Xeon Phi KNL, considerando diferentes optimizaciones aplicables.
- Se medirá el rendimiento considerando diferentes escenarios en cada caso y se realizará un análisis de estos.
- Se comparará la propuesta llevada a cabo y sus resultados con otros existentes en la literatura.

1.3. Contribuciones

Las contribuciones de esta tesina son las siguientes:

- Una implementación optimizada para computar los caminos mínimos en un grafo de acuerdo con el método FW sobre aceleradores Xeon Phi KNL. Como se mencionó en la sección 1.1, esta implementación podría beneficiar a aplicaciones de ámbitos diversos como el tráfico automovilístico, las redes sociales, las redes de computadoras, la bioinformática, entre otros. Para beneficio de la comunidad científica, esta implementación se encuentra disponible en un repositorio web público [https://github.com/ulisescosti/Tesina-FW-XeonPhiKNL].
- Una evaluación de rendimiento de aceleradores Xeon Phi KNL para computar los caminos mínimos en un grafo de acuerdo con el método FW. Este análisis no sólo permite evidenciar qué optimizaciones tienen mayor impacto en la mejora de rendimiento sino también apreciar el potencial de esta arquitectura para el problema de estudio.

1.4. Publicaciones

Esta tesina ha servido como base para la siguiente publicación científica:

- "*Comparación de Arquitecturas HPC para Computar Caminos Mínimos en Grafos. Intel Xeon Phi KNL vs NVIDIA Pascal*", M. Costanzo, E. Rucci, U. Costi, F. Chichizola, y M. Naiouf. En: "*Actas del XXVI Congreso Argentino de Ciencias de la Computación (CACIC 2020)"* (*En prensa*).

1.5. Organización

En el capítulo 2 se detalla al procesador Xeon Phi KNL, incluyendo su propósito, su arquitectura con la conformación básica de sus núcleos, los conjuntos de instrucciones soportados, así como también los diferentes modos de ejecución y de memoria que provee. Luego se presentan los modelos de programación utilizados en el KNL; y finalmente se detalla una lista de optimizaciones generales que el software debe cumplir si se pretende extraer el mayor rendimiento posible de este procesador.

El capítulo 3 detalla el problema de caminos mínimos de grafos, el algoritmo FW y sus variantes orientadas a la computación paralela.

En la capítulo 4 se presentan las distintas versiones de FW desarrolladas para la arquitectura del Xeon Phi KNL, describiendo cada optimización aplicada y presentando los resultados de rendimiento en las tablas y gráficos correspondientes.

Por último, en el capítulo 5 se presentan las conclusiones y trabajos futuros.

Capítulo 2

Intel Xeon Phi Knights Landing

En este capítulo se comienza presentando la organización y arquitectura del procesador (sección 2.1). Luego se enuncian los modelos de programación (sección 2.2), optimizaciones de software para aprovechar el potencial del procesador (sección 2.3), y por último, un resumen del capítulo (sección 2.4).

2.1. Organización y arquitectura

El procesador Intel Xeon Phi Knights Landing (KNL) es un procesador del tipo *many-core*, es decir, a diferencia de un *multi-core* tradicional, éste dispone de núcleos más simples pero en una cantidad mucho mayor. Está orientado y optimizado para un alto nivel de paralelismo explícito, a expensas de proveer menor rendimiento en la ejecución de un único hilo.

A diferencia de otros tipos de procesadores *many-core y* GPUs, los Xeon Phi tienen una ventaja fundamental al mantener la arquitectura y conjunto de instrucciones x86. Ésto permite facilitar el desarrollo de software al permitir utilizar las herramientas de paralelización pre-existentes, tales como OpenMP y MPI, y siendo compatibles con cualquier librería y ejecutable x86 preexistente. El objetivo del Xeon Phi es competir en el área de HPC al proveer una potencia de cómputo de paralelismo masivo comparable al de una GPU de alto rendimiento, pero reduciendo el esfuerzo de programación casi como si se tratase de un *multi-core* tradicional.

2.1.1. Origen

El Xeon Phi KNL tiene una arquitectura del tipo Intel Many Integrated Core (MIC), la cual nació siguiendo la filosofía de Larrabee, una arquitectura experimental de GPUs de Intel basada en x86. Si bien Larrabee tenía inicialmente planificado su primer producto comercial para ser lanzado 2010, finalmente esto no se cumplió. Todo el proyecto terminó siendo cancelado debido a continuos retrasos en el desarrollo y por resultados iniciales de rendimiento muy por debajo de los esperados. No obstante, Larrabee sirvió de base para desarrollos futuros de otras arquitecturas del tipo *many-core* como es el caso de MIC.

La arquitectura MIC reutilizó gran parte de la investigación de Larrabee, heredando de ésta elementos tales como las unidades de procesamiento vectorial (unidades SIMD) de 512 bits, y el sistema de coherencia de caché, entre otros. La principal diferencia con Larrabee, es que un

procesador de arquitectura MIC ya no es más una GPU, sino que se trata más bien de un coprocesador orientado fuertemente a HPC.

El primer procesador comercial de arquitectura MIC fue el Intel Xeon Phi - Knights Corner (KNC), lanzado en 2013. Éste, al igual que su predecesora arquitectura Larrabee, tiene a sus núcleos basados en la arquitectura P5 (la de los primeros Pentium de los años 90), es decir, se trata de núcleos superescalares con ejecución en-orden. También se mantuvo el agregado de una unidad SIMD de 512 bits, la implementación de Hyper-Threading de 4-vías (capaz de ejecutar simultáneamente cuatro hilos por núcleo), y la extensión de la arquitectura de x86 a x86-64. El procesador, según el modelo, posee entre 57 y 61 núcleos, interconectados por un bus en anillo, y funcionan a una frecuencia de entre 1053 y 1333MHz. Con Hyper-Threading activado, un modelo de 61 núcleos es capaz de ejecutar 244 hilos en simultáneo. Todos los Xeon Phi KNC salieron al mercado como coprocesadores en formato de tarjeta aceleradora, en su mayoría con interfaz PCI-Express 2.0 16x. Según el modelo en particular, el KNC posee entre 6 y 16GB de memoria GDDR5 (memoria de gran ancho de banda, común en tarjetas gráficas).

En 2016, con el avance en el proceso de litografía de 22nm a 14nm, se lanzó un nuevo Xeon Phi, llamado Knights Landing (KNL), el cuál no solo elevó la cantidad de transistores de 5 mil millones a 8 mil millones (manteniendo un tamaño similar de pastilla), sino que además se implementaron cambios profundos en la arquitectura con el objetivo de aumentar fuertemente el rendimiento *single-thread*.

2.1.2. Novedades de KNL con respecto a las versiones anteriores de Xeon Phi

Si bien con Knights Landing casi se duplicó la cantidad de transistores con respecto a Knights Corner, esto no se tradujo a duplicar también la cantidad de núcleos; ya que solo pasaron de 61 (en *KNC*) a 72 con KNL en los respectivos modelos tope de gama. Esto se debe a que los diseñadores de la arquitectura más bien priorizaron incrementar el valor de IPC*[1], a modo de poder cumplir el objetivo de mejorar el rendimiento *single-thread* (mono-hilo). En otras palabras, los circuitos extra provistos por el salto a los 14nm, fueron utilizados en mayor medida para incrementar la complejidad y potencia de los núcleos mas que para aumentar la cantidad de los mismos.

Para lograr el objetivo de incrementar sustancialmente el rendimiento *single-thread*, se efectuó un fuerte rediseño de la arquitectura a nivel núcleo. Éstos, ahora están basados en la arquitectura Silvermont en lugar de la P5, por lo que se trata de núcleos mucho más complejos, que implementan ejecución fuera de orden de ancho 2 (se traduce en más IPC). Ésto, sumado al agregado de una segunda unidad de procesamiento SIMD de 512 bits, y al aumento de

1 IPC = Instructions per cycle (Instrucciones por ciclo). Define la cantidad de instrucciones de un hilo que el procesador es capaz de ejecutar en paralelo en cada ciclo de reloj. El IPC a fines prácticos le da “un peso” al valor de frecuencia del procesador, que es el que determina la cantidad de ciclos por segundo que tiene su reloj. Por ejemplo, un procesador con IPC alto y frecuencia baja, podría en muchos casos ser más rápido que otro procesador que funciona a altas frecuencias, pero con un IPC mucho más bajo. Un ejemplo muy famoso de esto es la rivalidad de los años 2000-2006 entre los procesadores Athlon 64 de AMD (relativa baja frecuencia, alto IPC) y los Pentium 4 de Intel (muy alta frecuencia para la época, pero muy bajo IPC).

frecuencia de 1333MHz (con Knights Corner) a 1700MHz, se logra al menos el doble de rendimiento *single-thread* con respecto a KNC.

Otra novedad de KNL es que no requiere de un procesador *host* como sí lo hacía KNC, ya que KNL es capaz de comportarse por sí mismo como un procesador *host* autónomo; es decir, es capaz de bootear sistemas operativos, conectarse directamente a una red (ya sea Ethernet, Infiniband, o similar), y no sufre del *overhead* del *off-loading* que sí está presente en los KNC y en las GPU, en los cuales se necesita inevitablemente transferir datos desde y hacia la placa PCI-Express. Inicialmente se iban a lanzar también modelos de KNL en formato PCI-Express como exclusivamente coprocesadores, pero fueron cancelados y solo salieron a la venta los modelos de procesadores autónomos. Los KNL, en lugar de venir con memoria GDDR5 (presente en los KNC), vienen con una memoria de 16GB de alto ancho de banda llamada MCDRAM, con sus chips integrados en la misma plaqueta del procesador. Además de la MCDRAM, el KNL (como cualquier otro procesador autónomo) utiliza directamente a la memoria DDR RAM, que en este caso, se trata de 6 canales de DDR4 que soportan un máximo de 384GB. Como resultado de todos estos cambios, el Xeon Phi KNL ahora ofrece picos de rendimiento para punto flotante de alrededor de 3 TFLOP/s con doble precisión y 6 TFLOP/s con simple precisión.

Otra característica de KNL es que, a diferencia de KNC, éste mantiene la compatibilidad con los sets de instrucciones de procesamiento vectorial *legacy* como las SSE, y las AVX*x*; e introduce AVX-512, un conjunto de instrucciones vectoriales de 512 bits que son arquitectónicamente consistentes con las anteriores AVX y AVX2 de 256 bits. KNC proveía también un conjunto de instrucciones vectoriales de 512 bits pero era uno específico del procesador, no pensado como una evolución natural de AVX2. Todas estas características de retrocompatibilidad de los sets de instrucciones, sumado a que KNL se trata de un procesador x86-64 capaz de funcionar como un procesador *host* autónomo, le permite brindar la compatibilidad binaria con programas previamente compilados no solo para procesadores Xeon estándar, sino que también es capaz de ejecutar cualquier programa o librería *legacy* compilados para las arquitecturas x86 o x86-64 genéricas.

2.1.3. Unidad de replicación escalable de la arquitectura

Organización en tiles

La unidad básica de replicación en la arquitectura del Xeon Phi KNL es el ***tile*** (“baldoza” / ”azulejo”). Como se muestra en la figura 2.2, un *tile* está conformado por dos núcleos, dos unidades de procesamiento vectorial por núcleo, y 1MB de caché L2 compartida entre ambos núcleos. Cada pastilla del KNL cuenta con 38 *tiles*, de los cuales a lo sumo 36 están activos (en los modelos tope de gama); los otros *tiles* restantes se desactivan por motivos de eficiencia en el proceso de fabricación. En la figura 2.1 se muestra un diagrama de bloques del paquete, donde se puede apreciar la distribución de los tiles a lo largo de todo el chip, los controladores de memoria, y los integrados de MCDRAM.

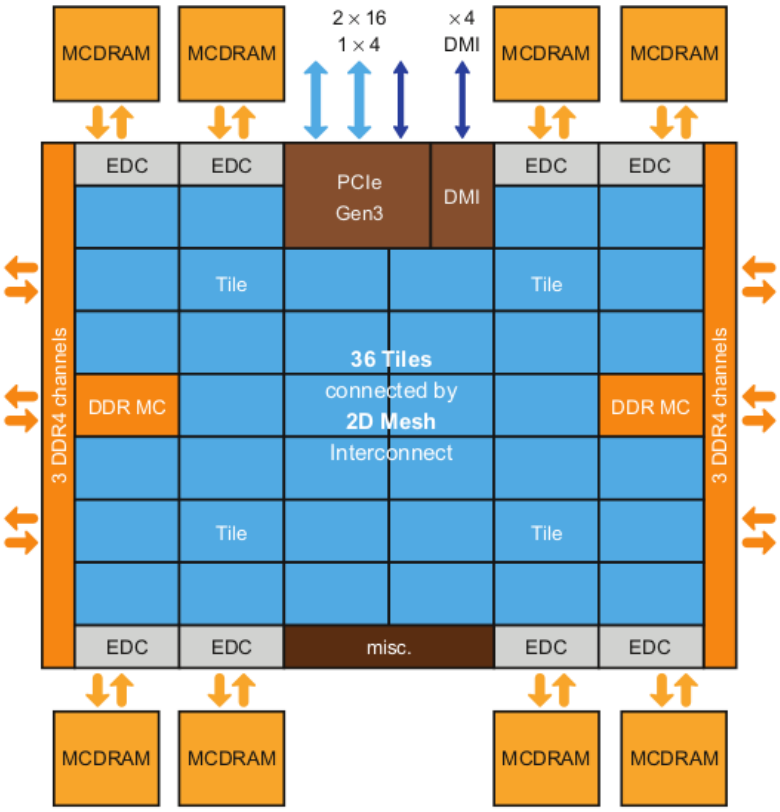

Fig 2.1: Diagrama de bloques del paquete del KNL (CPU).

Fuente: FIG 2.2 [8, p. 18]

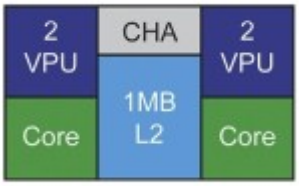

Fig 2.2: Diagrama de bloques de un tile.

Fuente: FIG 2.3 [8, p. 18]

Interconexión de tiles

Los *tiles* se encuentran interconectados por una red de interconexión de tipo *mesh* (malla) 2D, que implementa coherencia de caché con directorio distribuido. La malla interconecta filas y columnas de *tiles* con un protocolo de ruteo estático, con el cual, cada transacción de memoria viaja primero verticalmente hasta llegar a la fila buscada, y luego viaja horizontalmente hasta llegar al tile destino. Este sistema es mucho más eficiente y escalable que el sistema de interconexión de anillo 1D de Knights Corner, por lo que provee un mayor ancho de banda y a la vez menor latencia.

2.1.4. Modos de ejecución (modos cluster)

Introducción

La malla que interconecta los tiles del KNL se puede configurar en tres modos de ejecución distintos: A*ll-to-all*, Q*uadrant*, y S*ub-NUMA Clustering (SNC)*. Estos modos particionan al procesador en regiones virtuales separadas estableciendo cierta afinidad de direcciones de memoria con la partición de la malla que le corresponde. Esto no es un impedimento para

mantener la coherencia de caché en todo el procesador a lo largo de todas las particiones, cosa que ocurre independientemente del modo cluster elegido.

El KNL implementa la coherencia de caché L2 con directorio distribuido y protocolo MESIF. El directorio se encuentra distribuido entre los *Cache/Home Agent* (CHA). Cada *tile* alberga uno de estos agentes de caché.

Es importante remarcar que cualquier software compilado para x86 puede ejecutarse en cualquiera de los modos cluster. No obstante, para ganar rendimiento aprovechando al máximo las características del KNL, normalmente es deseable efectuar modificaciones al código si no fue diseñado puntualmente para esta arquitectura. La novedad es que no solo el software es el que puede modificarse para aprovechar las características del procesador, sino que con KNL, el procesador también puede adaptarse al software al poder elegir manualmente el modo cluster a utilizar según las características del software que se desea ejecutar.

Modo All-to-all

Es el modo cluster más lento, pero el único disponible cuando los módulos de memora DDR no son idénticos en capacidad. En este modo no hay afinidad entre los *tiles*, agentes de caché y controladores de memoria. Esta carencia de afinidades es lo que brinda la flexibilidad necesaria para mantener la compatibilidad con canales DDR asimétricos. No obstante, con este esquema se pierde rendimiento, ya que las transacciones por fallo de caché L2 recorren distancias más largas en promedio, y por consiguiente, pueden tardar más tiempo en resolverse. La única ventaja de este modo cluster sobre el resto es que soporta cualquier configuración de módulos DDR.

Modo Quadrant

Es el modo cluster por defecto, pero requiere que se instalen módulos DDR de la misma capacidad. Si se detecta una asimetría en los canales de memoria, el modo *All-to-all* será seleccionado automáticamente.

Con el modo *Quadrant*, el chip queda dividido lógicamente en cuatro cuadrantes, y se establece una afinidad entre los agentes de caché y los controladores de memoria. Esto hace que los fallos de caché L2 tarden menos tiempo en resolverse en comparación al que requerirían en el modo *All-to-all*.

Modo SNC-4

El modo SNC implementa una afinidad entre los tres elementos vinculados en las transacciones en los fallos de caché L2; es decir, el *tile*, el agente de caché y el controlador de memoria. Esto permite dividir al chip en cuatro cuadrantes como cuatro clusters NUMA*[1]. En este modo, ante un fallo de caché L2, si la posición de memoria se encuentra en el mismo cuadrante, queda

1 NUMA es un diseño de memoria, en el cual a diferencia del esquema UMA tradicional, los accesos tienen determinísticamente un tiempo de acceso mejor o peor según en qué zona de memoria se encuentre el dato a operar; por lo que el software que corre en un sistema NUMA debe hacer un uso discrecional de las zonas rápidas y lentas de la memoria para lograr tiempos de acceso óptimos.

garantizado que el fallo se resolverá internamente en el mismo cuadrante, logrando así los tiempos de acceso más óptimos de los tres modos cluster.

Si el software no está optimizado para NUMA y accede frecuentemente a zonas de memoria "lejanas" (pertenecientes a otros cuadrantes), muy probablemente se obtenga menos rendimiento con SNC-4 que con *Quadrant*.

Modo Hemisphere y SNC-2

Estos modos son simples variaciones de los modos *Quadrant* y SNC-4 respectivamente. En *Hemisphere* en lugar de dividir el chip en cuatro cuadrantes, se divide en dos hemisferios. De igual manera, con SNC-2 el sistema se divide en dos clusters NUMA en lugar de cuatro. En estos modos se obtiene menos rendimiento que con *Quadrant* y SNC-4, pero siguen teniendo menor latencia y mayor ancho de banda que el modo *All-to-all*.

2.1.5. Tipos de memoria (MCDRAM y DDR)

Memoria DDR

Al ser el Xeon Phi KNL un procesador autónomo, como cualquier *multi-core* tradicional, puede acceder a la memoria DDR de gran capacidad. El KNL soporta 6 canales DDR4, con un ancho de banda combinado de aproximadamente 90 GB/s, y con una capacidad máxima total de 384GB, dependiendo del tamaño de los módulos instalados.

Memoria MCDRAM

Además de la memoria DDR, el procesador KNL cuenta con una memoria de 16GB de alto ancho de banda llamada MCDRAM. Esta memoria complementaria le brinda al KNL la posibilidad de tener un ancho de banda similar al de las GPUs, sin tener que sacrificar capacidad para lograrlo (ventaja clave de poder acceder a ambas memorias).

Los chips de la MCDRAM (de 2 GB cada uno en este caso) están integrados en el paquete del procesador; es decir, están soldados en la misma plaqueta; y el ancho de banda combinado de los ocho chips de MCDRAM es superior a los 450GB/s. Estas diferencias técnicas se presentan en la tabla 2.1.

Tabla comparativa

	Capacidad	**Ancho de banda**	**Latencia en idle**
MCDRAM	16GB	Hasta 450GB/s	~ 150ns
DDR	Hasta 384GB	Hasta 90GB/s	~ 125ns

Tabla 2.1: Tabla comparativa de memorias MCDRAM y DDR.

Fuente: FIG 4.15 [8, p. 82]

2.1.6. Modos de memoria

La memoria puede ser configurada en tiempo de booteo en uno de los siguientes modos:

I. Modo *Cache*: la MCDRAM se comporta como una caché de la DDR.

II. Modo *Flat* (plano): la MCDRAM es tratada como una memoria estándar en el mismo espacio de direcciones de la DDR.

III. Modo *Hybrid* (híbrido): una porción de la MCDRAM funciona como caché de la DDR, y la porción restante queda en la modalidad *flat*.

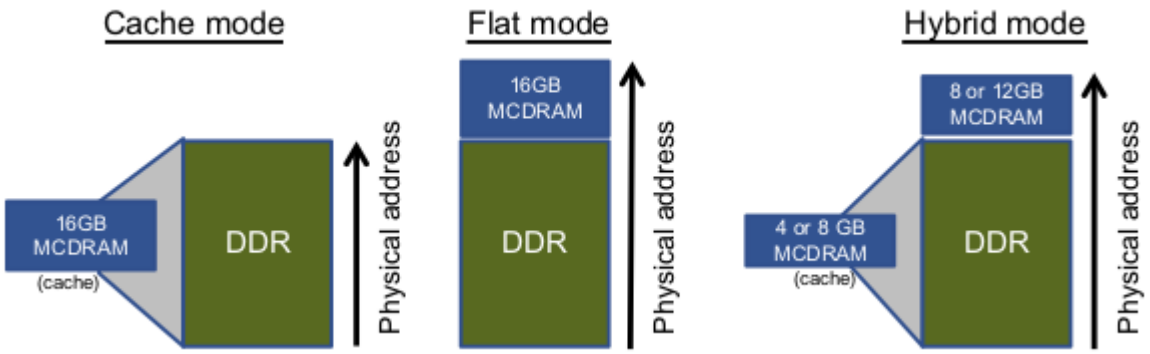

Fig 2.3: Gráfico representativo de los tres modos de memoria del KNL.

Fuente: FIG 4.13 [8, p. 79]

A continuación se describen los modos de memoria más en detalle:

Modo Cache

En este modo, la memoria MCDRAM es configurada y administrada por hardware como una caché de la memoria DDR. Es el modo de memoria ideal para ejecutar software *legacy*, ya que le permite gozar del ancho de banda de la MCDRAM mientras ésta es usada de forma transparente, sin tener que solicitarla por ningún medio. En este modo, el software utiliza la DDR como si se tratase de la única disponible, y es el sistema de memoria el que se encarga de cachearla con la MCDRAM automáticamente, lo que efectivamente le brinda la transparencia requerida. Para gozar de los beneficios de este modo, no se requiere de ninguna modificación en el código fuente ni ninguna modificación en los parámetros del ejecutable. Para la mayoría de las aplicaciones, este será el modo de memoria más apropiado. No obstante, para algunas aplicaciones con naturaleza “poco amigable” con las memorias caché (es decir, con baja tasa de *hits*), probablemente puedan obtener mayor rendimiento con los otros dos modos de memoria.

Modo Flat

El modo *flat* permite que el sistema operativo vea a la memoria MCDRAM como otra memoria direccionable, es decir, como un nodo NUMA separado. Esto permite que por software, selectivamente, se pueden utilizar a ambas memorias (DDR y MCDRAM) haciendo uso explícito de ellas. Por defecto, todos los datos se ubicarán en la DDR, reservando a la MCDRAM como un recurso preciado, para que el programador ubique en la misma solo los datos más críticos con respecto a la demanda de ancho de banda. El resultado, es que si el programador utiliza la MCDRAM correctamente, el rendimiento será más consistente que el obtenido con el modo caché, al no depender de ninguna tasa de *hits*.

Para elegir el tipo de memoria a utilizar hay disponibles diversos métodos y herramientas. Una de las herramientas más comunes utilizada para manejar la MCDRAM es la librería Memkind, la cual permite definir explícitamente en el código (de forma bastante amigable) cuándo reservar espacio en memoria DDR y cuándo en la MCDRAM. No obstante, para mantener la portabilidad del software *legacy*, también se puede definir el tipo de uso que se le dará a la memoria MCDRAM antes de la carga de un programa; permitiendo así que dicho programa legacy pueda hacer un uso directo de la memoria MCDRAM en modo *flat* sin ser "consciente" de ello.

Modo hybrid

En el modo *hybrid*, una parte de la memoria MCDRAM se utiliza como caché y la otra parte se utiliza como memoria *flat*, por lo que se trata de una combinación entre ambos modos de memoria. En este modo, la parte de la MCDRAM que se utiliza como *flat* queda en un nodo NUMA separado, mientras que la parte restante queda como caché de toda la memoria DDR.

Este modo es útil para aplicaciones que se benefician del *caching* gracias a una alta tasa de *hits*, y que a su vez pueden potencialmente mejorar aún más su rendimiento al utilizar selectivamente la MCDRAM.

2.1.7. Simultaneous Multi Threading (SMT)

El Xeon Phi KNL implementa la tecnología SMT bajo el nombre comercial Hyper-Threading, la cual le permite ejecutar múltiples hilos simultáneamente en un mismo núcleo. Esta tecnología se aprovecha de las arquitecturas superescalares de los procesadores, para que estos en vez de captar siempre instrucciones de un mismo hilo, ahora lo puedan hacer con instrucciones de distintos hilos simultáneamente, sin que esto implique ningún tipo de *context switch*. El hardware extra requerido para que un procesador implemente esta tecnología es mínimo, por lo que básicamente, SMT permite explotar los recursos preexistentes de los procesadores, algo que en muchos casos no es posible de lograr con la ejecución de un único hilo por núcleo.

La mejora de rendimiento obtenida con el uso de SMT es algo dependiente de las características de la aplicación a correr, pudiendo obtener mejoras de rendimiento del orden del 40%, no obtener mejora alguna, o incluso tener un impacto negativo sustancial en el rendimiento.

Si bien la tecnología SMT no aumenta la cantidad real de núcleos, los sistemas operativos ven a un procesador con SMT como si tuviera una cantidad de núcleos igual a la cantidad de hilos que es capaz de ejecutar en simultaneo. Por ejemplo, a un procesador de 8 núcleos con SMT de dos hilos por núcleo, los sistemas operativos lo verán como un procesador de 16 núcleos. Aquí entran en juego las distinciones “núcleo lógico” y “núcleo físico”, utilizadas frecuentemente para diferenciar los núcleos que el sistema operativo ve (núcleos lógicos) con los núcleos reales del procesador (núcleos físicos).

2.2. Modelos de programación

Los modelos de programación soportados por el KNL varían según si éste es utilizado como procesador nativo o como coprocesador. No obstante, en esta tesina se hará hincapié en la configuración del KNL como procesador nativo, ya que se trata del modo más ampliamente utilizado en general, y es el que será empleado a lo largo del desarrollo de este trabajo.

Un KNL como procesador nativo soporta los mismos modelos de programación disponibles para cualquier *multi-core* del mercado, ya sea OpenMP, MPI*[1], o el híbrido MPI+X (por ejemplo, MPI + OpenMP), siendo en general cada uno de estos más afín a un determinado modo cluster y modo de memoria. En este trabajo, el modelo de programación a utilizar será el de OpenMP, por lo que de aquí en adelante se hará principal énfasis en éste último.

2.2.1. OpenMP

OpenMP es un estándar para la programación paralela en memoria compartida. Se compone de un conjunto de directivas, un conjunto de variables de entorno y un conjunto de funciones de librería. Su origen radica en la necesidad de poder desarrollar programas paralelos escalables a un bajo costo de programación. En ese sentido, se dice que OpenMP sigue una filosofía de diseño incremental, ya que no requiere escribir un código “desde cero”.

OpenMP se basa en el modelo *Fork-Join*. Un programa paralelo implementado con OpenMP arranca con un hilo maestro, y éste al llegar al primer bloque paralelo crea los correspondientes hilos *workers* (*Fork*). Estos hilos trabajarán concurrentemente dentro del bloque paralelo, y en el caso en que haya un *for* paralelo (o una sección) repartirá la carga del mismo entre ellos. Al final de cada *for* paralelo (o sección paralela) hay una barrera implícita, donde cada hilo se duerme esperando a que el resto termine su trabajo. Luego, vuelven a repartirse la carga en caso de que haya consecutivamente otro *for* o sección paralela. Una vez que se termina la ejecución de todo el bloque paralelo, se eliminan los hilos *workers* (*Join*), para luego continuar la ejecución del hilo maestro en solitario. Cuando el hilo maestro llega otra vez a un bloque paralelo, nuevamente se vuelven a crear los hilos *workers* necesarios (*Fork*), y otra vez se cumple el mismo ciclo descrito anteriormente por cada bloque paralelo encontrado en el programa hasta su finalización.

1 MPI: es un estándar de pasaje de mensajes para programas paralelos distribuidos.

Directivas básicas

A continuación se listan algunas de las directivas de OpenMP*[1] para C y C++ más básicas utilizadas a lo largo de este trabajo.

#pragma omp parallel

- Define un bloque de código que se va a ejecutar en paralelo.
- Se ubica una línea antes del comienzo del bloque.
- Cuando el bloque se ejecuta, al comienzo se crean los hilos en la cantidad definida en la variable de entorno OMP_NUM_THREADS. Al finalizar la ejecución del bloque se matan dichos hilos.

#pragma omp for

- Se utiliza dentro de un bloque *parallel*, ubicándose una linea antes del comienzo de un *loop* (*for* en C y C++).
- Indica al compilador que debe paralelizar (a nivel de *threads*) a un *loop* determinado. Si hay *loops* anidados dentro, estos se ejecutaran secuencialmente dentro del contexto de cada hilo.
- Reparte las iteraciones del *loop* entre los distintos hilos correspondientes al bloque *parallel*. Esta repartición de trabajo entre hilos por defecto es estática, pero puede hacerse dinámica, con distintas sub-variantes según ciertos parámetros opcionales de la directiva.
- Al final del *loop* paralelizado, por defecto se deja un *join* (barrera) implícito.

Cláusulas de visibilidad de datos

private(<lista de variables>)

Las variables definidas en esta cláusula se guardan en la memoria *stack* local de cada hilo, por lo que estos llevan una copia local de cada una de las variables. Es la cláusula por defecto para la variable utilizada como índice de un *loop* paralelizado.

firstprivate(<lista de variables>)

Similar a *private*, pero las variables primero se inicializan con el valor definido previamente en el hilo maestro.

shared(<lista de variables>)

Cláusula por defecto de todas las variables (excepto la variable usada como índice de un *loop* paralelizado), estas permanecerán en el contexto global, por lo que serán utilizadas de forma compartida por todos los hilos del bloque paralelo.

1 Para más información sobre las diferentes directivas de OpenMP: [https://www.openmp.org]

2.3. Optimizaciones

Es importante remarcar que si se parte de una aplicación no optimizada para la ejecución paralela, centrarse desde el "minuto cero" en la optimización fina para la arquitectura del KNL no suele ser lo ideal. Generalmente resulta más efectivo comenzar con las técnicas de paralelización y optimización genéricas para *multi-cores*; y por lo tanto, demás está decir que también se deben tener en cuenta las optimizaciones respectivas desde el punto de visa de la ejecución individual de un hilo. Habiendo dicho esto, ahora partiendo de una aplicación optimizada para *multi-cores* en general, se puede empezar a introducirse de a poco en las particularidades mas finas de la arquitectura del KNL. Primero se puede comenzar eligiendo el modo cluster y el modo de memoria más apropiados según las características del problema, como se verá a continuación.

2.3.1. Modos Cluster

Se puede dividir a los modos cluster en dos grupos según cómo los diferentes modelos de programación se adecúan a ellos:

Modos SNC-4 y SNC-2

Con los modos SNC es posible obtener más rendimiento que con *Quadrant* en aplicaciones del tipo "*NUMA aware*"; es decir, aplicaciones ya optimizadas para NUMA. Estas aplicaciones tienen un gran potencial de lograr mejor rendimiento con estos modos cluster, gracias a utilizar discrecionalmente los accesos a memoria "cercanos" y "lejanos", buscando así maximizar las comunicaciones internas (accesos "cercanos") y minimizando las comunicaciones entre distintos clusters NUMA (accesos "lejanos").

Es importante remarcar que para un núcleo, lo que determina si un dato va a estar en "memoria cercana" o en "memoria lejana" con respecto a él, es si fue desde ese mismo núcleo (o desde otro de su mismo cuadrante) donde se accedió por primera vez al dato en cuestión. Si el dato fue accedido primero por un núcleo perteneciente otro cuadrante diferente, entonces dicho dato quedará en "memoria lejana" desde su punto de vista. Por este motivo, es muy importante analizar cómo las aplicaciones inicializan sus datos. Se deben evitar comportamientos como el de un proceso/hilo maestro que se encarga de inicializar las estructuras en memoria para todos los hilos de los diferentes cuadrantes; esto es algo que no debe suceder si se espera obtener buen rendimiento con los modos cluster SNC. También puede ser perjudicial para el rendimiento en los modos cluster SNC si un hilo es cambiado de núcleo por el sistema operativo y termina en un núcleo de un cuadrante distinto, por lo que toda su memoria "cercana" pasaría inmediatamente a estar en memoria "lejana" desde su punto de vista. Esto se soluciona estableciendo una afinidad de hilos con núcleos, como se verá más adelante en la sección 2.3.4.

Las aplicaciones del tipo MPI / MPI+X que utilizan 4 procesos por nodo, o bien un múltiplo de 4 procesos por nodo, tienen naturalmente grandes probabilidades de obtener mejor rendimiento con SNC-4 que con el resto de los modos cluster. Ademas de coincidir (o ser múltiplo) la cantidad de procesos por nodo con la cantidad de clusters NUMA que habría disponibles con

SNC-4, se le suma el hecho de que normalmente este tipo de aplicaciones suelen estar pensadas desde cero con la correspondiente minimización de la comunicación entre procesos (accesos lejanos), en favor de las comunicaciones internas entre hilos de un mismo proceso (accesos cercanos). Del mismo modo (y por descarte), las aplicaciones con 2 procesos por nodo, o un múltiplo de 2 procesos por nodo, tienen chances de ejecutarse más rápido con SNC-2 que con el modo *Quadrant*.

Para aplicaciones MPI que no tienen una cantidad adecuada de procesos por nodo y/o que no localizan suficientemente las comunicaciones, probablemente corran mejor con los modos *Quadrant* o *Hemisphere*.

Modos Quadrant, Hemisphere y All-to-all

Desde el punto de vista del software, con estos modos cluster, el KNL se comporta como si se tratase de un único *multi-core*. De hecho, un programa implementado con cualquier modelo de programación utilizado para *multi-cores*, puede correr en el KNL sin más modificaciones, y aún así esperar tener buenos resultados de rendimiento. Estos tres modos cluster son transparentes desde el punto de vista del software; por lo que para una aplicación, lo único que puede variar es el rendimiento obtenido con cada uno de ellos.

2.3.2. Modos de memoria

La decisión entre elegir el modo de memoria *cache* o el modo *flat*, dependerá principalmente de si las aplicaciones a correr son "*cache-friendly*" o no. Si una aplicación ya de por sí es de naturaleza "*cache-friendly*", entonces probablemente corra mejor con el modo de memoria *cache*; mientras que las aplicaciones restantes probablemente puedan obtener más rendimiento con los modos *flat* y *hybrid*.

2.3.3. Elección de cantidad de hilos por núcleo (SMT)

Si bien la estrategia de utilizar la mayor cantidad de núcleos posibles suele ser la ideal para el rendimiento, una vez que todos ellos tienen un hilo asignado, al agregarle más podría no mejorar el tiempo de ejecución, e inclusive hasta podría empeorarlo. Esto es algo muy dependiente de la aplicación, su granularidad y la "densidad" de accesos a memoria que efectúe cada hilo. No hay una receta maestra que determine la cantidad de hilos por núcleo óptima, por lo que al final de cuentas lo mejor es parametrizar dicha cantidad de hilos a utilizar, para luego probarlas empíricamente hasta dar con la óptima para una aplicación determinada.

Es esperable que una configuración de 3 hilos por núcleo no sea la más adecuada; en muy pocos casos con ésta se obtiene más rendimiento que con el resto de las configuraciones. Esto se debe a que en el KNL los recursos del núcleo (que se comparten dinámicamente entre los hilos) se pueden dividir en dos o en cuatro, pero no en tres. Es por este motivo que cada hilo bajo una configuración de tres hilos por núcleo utilizará un cuarto de los recursos del núcleo, en lugar de un tercio como intuitivamente uno podría creer [8, p. 108].

2.3.4. Afinidad de hilos con núcleos

El rendimiento de una aplicación puede variar sensiblemente según cómo sus hilos se distribuyen entre los núcleos de los procesadores. Una aplicación paralela habitualmente variará su rendimiento según cuán bien su distribución de hilos aproveche los recursos de cómputo del procesador, así como también según cómo dicha distribución de hilos hace uso de los distintos niveles compartidos de memoria caché. A continuación se detallarán ambos escenarios, enumerándolos como A y B, los cuales serán referenciados luego:

a) Aprovechamiento óptimo de los recursos de cómputo: si una aplicación trabaja con menos hilos que núcleos físicos disponibles, es deseable que dichos hilos se distribuyan de tal manera que la mayor cantidad posible de núcleos físicos tenga al menos un hilo que ejecutar*[1].

b) Aprovechamiento de la memoria caché: los hilos que comparten datos entre sí probablemente se verán beneficiados si se ejecutan en núcleos lógicos pertenecientes al mismo núcleo físico (comparten caché L1), o bien en núcleos físicos distintos pero del mismo *tile* (comparten caché L2 en el caso del KNL).

Si bien la estrategia A en general tiene mucho más peso en el rendimiento que la B, esta última también se toma en consideración en diversas configuraciones, como se verá más adelante.

El sistema operativo es el que por defecto se encarga totalmente de distribuir los hilos entre los núcleos del procesador, aunque este ligamiento también puede definirse manualmente. Asimismo, es posible utilizar herramientas más amigables que hacen este trabajo por nosotros, permitiendo elegir el patrón de distribución de hilos con unos simples parámetros, y luego la herramienta se encarga de implementar el ligamiento de forma transparente al usuario. Existen múltiples herramientas que brindan este tipo de soluciones. No obstante, en este documento se va a hacer principal énfasis en la Interfaz de Afinidad de Hilos de Intel, la cual será la herramienta a utilizar a lo largo de este trabajo.

Interfaz de Afinidad de Hilos (Intel)

Esta herramienta provee la variable de entorno KMP_AFFINITY, la cual permite pasarle una aglomeración de parámetros (previo a la ejecución de un programa), para que la interfaz se encargue de efectuar el ligamiento de hilos con núcleos siguiendo el patrón que el usuario (programador) desee. El principal parámetro a utilizar es el tipo de afinidad, el cual soporta distintos valores enumerativos, siendo los más relevantes: *balanced*, *scatter* y *compact*.

Tipos de afinidad

***scatter*:** lleva la estrategia A al máximo, ubicando los hilos de la forma mas dispersa posible a lo largo de todo el sistema. Suele ser la mejor opción para plataformas *multi-socket*. En la figura 2.6 se presenta un ejemplo de uso de este tipo de afinidad para una configuración de 8 hilos, en un hipotético Xeon Phi KNL de solo dos *tiles*.

1 Recordar que los recursos de cómputo se comparten entre los hilos de un mismo núcleo físico; por lo que dejar núcleos físicos enteros sin utilizar, representa un importante desaprovechamiento de recursos de hardware.

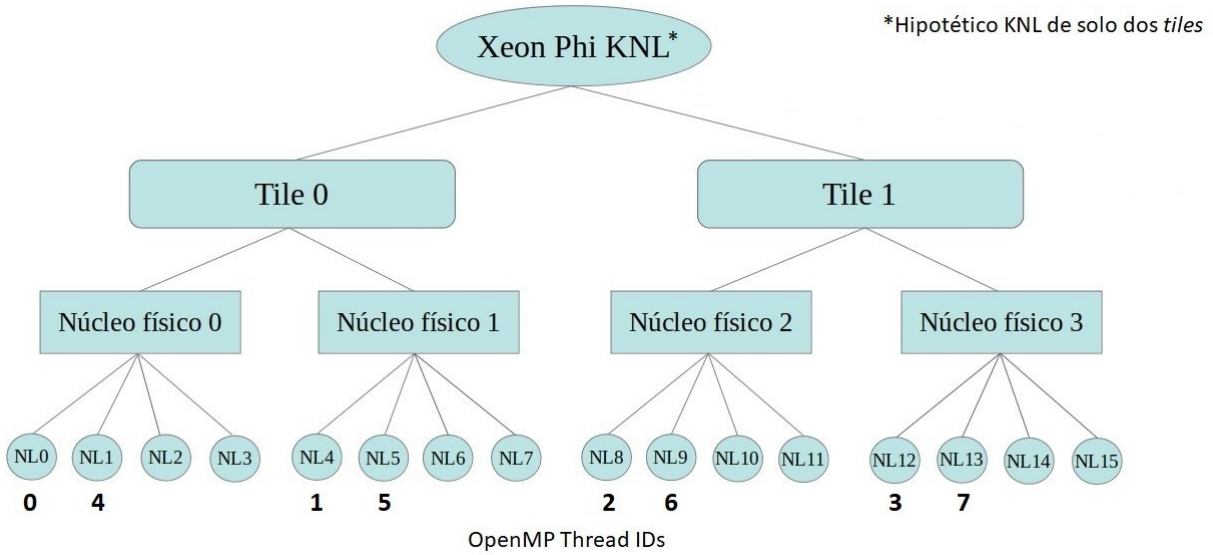

Fig 2.4: Ejemplo de distribución de hilos con tipo de afinidad "scatter"

***compact*:** Lleva la estrategia B al máximo, ubicando los hilos con ID*[1] consecutivos de forma ordenada entre núcleos lógicos consecutivos. De esta forma, se maximiza el uso compartido colaborativo de los diferentes niveles de caché. No obstante, este tipo de afinidad tiene fuertes perdidas de rendimiento si la cantidad total de hilos no alcanza para dar trabajo a todos los núcleos físicos del procesador. En la figura 2.5 se ejemplifica dicho escenario, donde la cantidad de hilos no es suficiente para aprovechar todos los núcleos físicos del procesador utilizando este tipo de afinidad.

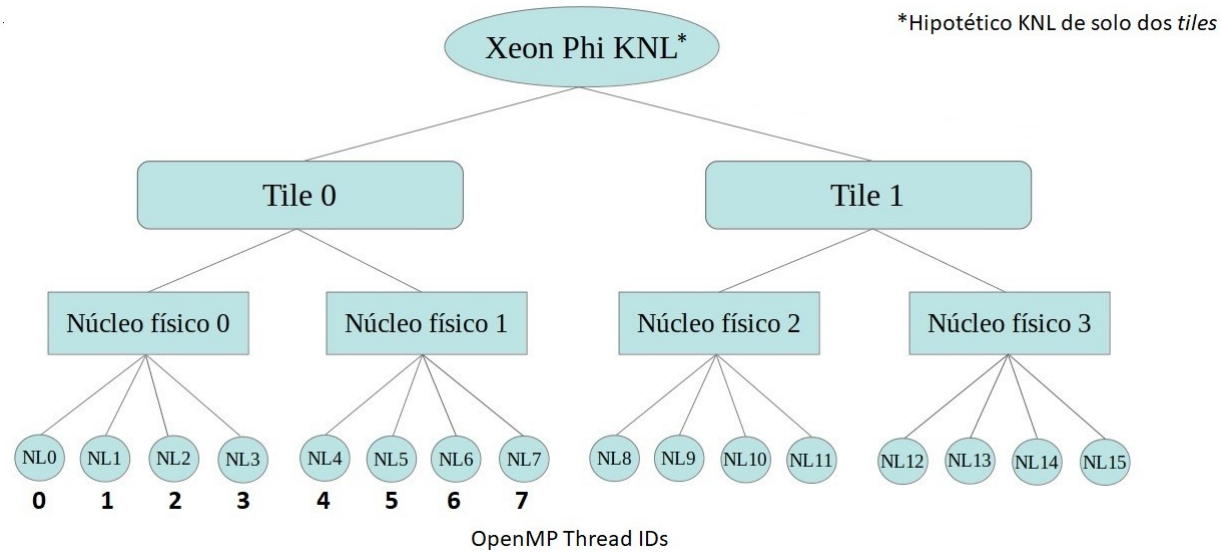

Fig 2.5: Ejemplo de distribución de hilos con afinidad "compact"

1 *Thread ID* de hilos de OpenMP en el contexto de OpenMP. Es decir, no se trata del *thread ID* de los hilos a nivel del kernel del sistema operativo.

balanced: es una solución híbrida que toma lo mejor de s*catter* y c*ompact*. Primero reparte los hilos de la forma más dispersa posible entre los distintos núcleos físicos hasta tener un hilo por núcleo físico (estrategia A). Luego asigna los restantes hilos a los núcleos lógicos que queden disponibles, de forma tal que los hilos que comparten cada núcleo físico sean de IDs consecutivos (estrategia B). Esta solución no está disponible para plataformas *multi-socket*. En la figura 2.7 se presenta un ejemplo con este tipo de afinidad.

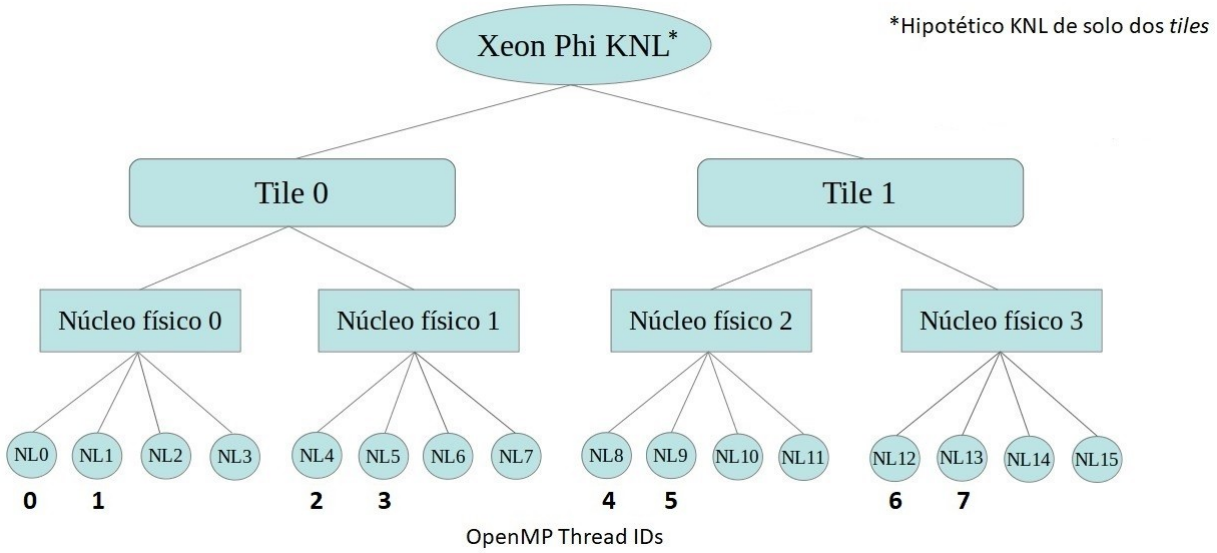

Fig 2.6: Ejemplo de distribución de hilos con afinidad "balanced"

Granularidad

thread/fine: cada hilo se liga individualmente con un núcleo lógico en particular, de modo que un hilo no puede cambiar de núcleo (ni físico ni lógico) en tiempo de ejecución. Los ejemplos de las figuras 2.5, 2.6 y 2.7 corresponden a este tipo de granularidad.

core: es la granularidad por defecto. En esta configuración, grupos de hilos se asocian a núcleos físicos, de modo que los hilos de cada grupo puedan “flotar” entre los núcleos lógicos que comparten el mismo núcleo físico. De esta forma, luego de un *context switch* de un hilo, su ejecución puede reanudarse inmediatamente en cualquiera de los otros tres núcleos lógicos “vecinos”. En la figura 2.7 se presenta un ejemplo de afinidad *balanced* con granularidad *core*.

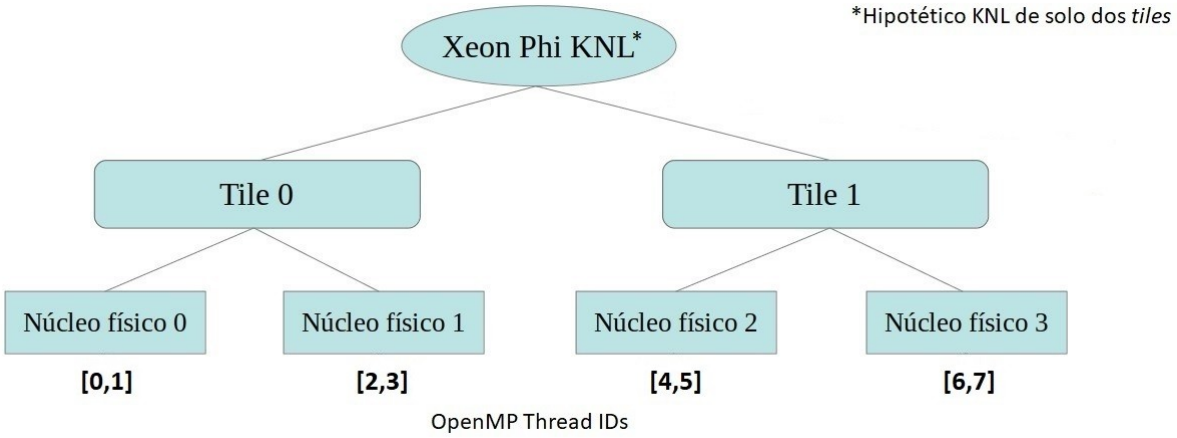

Fig 2.7: Ejemplo de distribución de hilos con granularidad "core" y tipo de afinidad "balanced"

tile: implementa la misma idea de la granularidad *core* pero llevada al nivel del *tile*. Es decir, liga grupos de hilos con *tiles*, de modo que los hilos de cada grupo puedan "flotar" entre los núcleos lógicos que comparten el mismo *tile*. Si bien un hilo luego de un *context switch* ahora tiene más oportunidades de reanudarse rápidamente (son el doble de núcleos lógicos en los que un hilo puede retomar el trabajo en comparación con la granularidad *core*), la desventaja es que si su ejecución se reanuda en el otro núcleo físico del mismo *tile*, su *working set**[1] habrá quedado en la caché L1D "equivocada", por lo que probablemente aumentará la tasa de fallos de caché hasta que se restablezca su *working set* en la L1D de su nuevo núcleo físico. En la figura 2.8 se muestra un ejemplo de afinidad *compact* con granularidad *tile*.

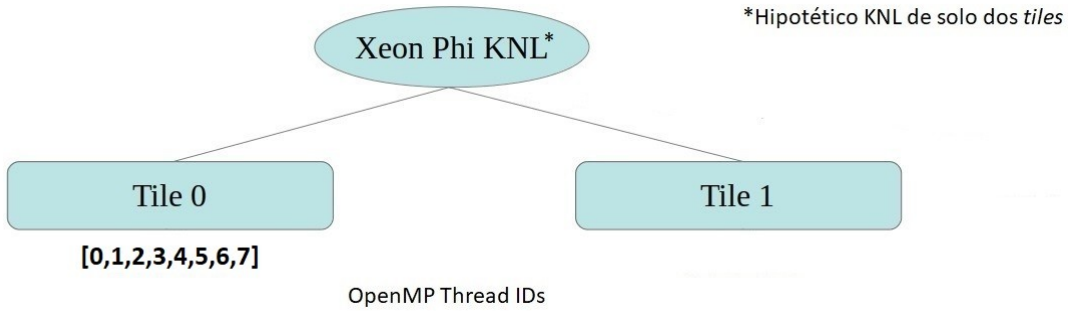

Fig 2.8: Ejemplo de distribución de hilos con granularidad "tile" y tipo de afinidad "compact"

1 El *working set* en su sentido más estricto es el conjunto de páginas de memoria más utilizadas de un proceso. En un sentido más laxo podemos hablar de *working set* como "espacio de memoria más utilizado", ya sea por un proceso (con todo su conjunto de hilos), o como es en este caso, por un hilo individualmente.

2.3.5. MCDRAM y DDR

Recomendaciones generales

Si bien el ancho de banda de la memoria MCDRAM es sustancialmente mayor que el de la DDR (ver tabla 2.1), la latencia de la MCDRAM es incluso algo más alta (es decir, peor) que la latencia de la DDR cuando no hay suficiente demanda de ancho de banda. Por lo que en algunas aplicaciones que tienen una localidad espacial y temporal deficientes, es probable que se obtenga el mismo (o incluso mayor) rendimiento si sus estructuras de datos son guardadas en la memoria DDR. En los casos en que no haya mejora de rendimiento al usar la MCDRAM, probablemente resulte más apropiado utilizar solamente la DDR, ya que ademas con ésta no hay que preocuparse de la limitación de los 16 GB de la MCDRAM.

En el caso en que se elija el modo *flat* o el modo *hybrid*, las estructuras de datos accedidas frecuentemente deberían ser alojadas en la memoria MCDRAM. Si dichas estructuras de datos superan la capacidad de la MCDRAM (16 GB en modo *flat*, y 12 u 8 GB en *hybrid*), entonces lo ideal sería colocar la parte más utilizada de ellas en la MCDRAM, y las menos utilizadas en la DDR. No obstante, el esfuerzo que requiere este tipo de implementación es alto, y puede no alcanzar para superar el rendimiento arrojado con el modo de memoria *cache*, motivo por el cual dicho modo de memoria es el más popular.

Modelos de uso de la memoria MCDRAM

Opción	Resumen
No hacer nada	Un programa en el cual su *working set* encaja perfectamente en la memoria L2 podría tener buen rendimiento con la DDR sola. Aplicaciones *cache-friendly* podrían seguir obteniendo ventajas con el modo de memoria *cache*. No obstante, para los programas que no pueden aprovechar bien la memoria caché, puede que obtengan algún beneficio de rendimiento si se selecciona el modo de memoria *flat* y se deja la MCDRAM sin utilizar.
Modo cache	Es trivial intentarlo. No implica ningún cambio en el código. En el KNL se puede dedicar el 100%, 50% o el 25% de la memoria MCDRAM como caché.
Las opciones restantes solo son útiles en los modos de memoria *flat* y *hybrid*	
numactl (NUMA Control utility)	Sin cambios en el código: Si está instalado el comando numactl, se lo puede utilizar para establecer previo a la ejecución de un programa, que todas las asignaciones de memoria se hagan en la MCDRAM (incluyendo *stacks* y segmentos de datos).
autohbw	Sin cambios en el código: Se puede utilizar la librería autohbw (parte del proyecto memkind) para que efectúe las asignaciones de memoria, desde un rango de direcciones determinado, a la MCDRAM.
memkind	Implica cambios en el código: Utilizando la librería memkind, se puede tener control por software de dónde se asigna memoria en cada caso, si en la DDR o en la MCDRAM.

Tabla 2.2: Modelos de uso de la memoria MCDRAM.

Fuente: Fig 3.2 [8, p. 29]

2.3.6. Procesamiento vectorial (SIMD)

Instrucciones SIMD en el Xeon Phi KNL

El Xeon Phi KNL es un procesador concebido desde cero con las unidades de procesamiento vectorial de 512 bits como pilares centrales de su poder de cómputo, relegando a un lugar más secundario a las unidades de ejecución de cómputo escalar. Por este motivo resulta esencial que una aplicación esté vectorizada en la mayor medida posible si se quiere sacar el máximo provecho al KNL.

Una aplicación diseñada y compilada para el KNL debería utilizar (siempre que sea posible) a los conjuntos de instrucciones AVX-512 (procesa más cantidad de datos en simultáneo), o en su defecto a los conjuntos AVX de 256 bits. Se debe evitar a toda costa el uso de instrucciones SSE y x87. Si bien esta arquitectura, con respecto a las anteriores, reduce en gran medida el impacto de mezclar instrucciones AVX con SSE, no obstante no logra anular del todo la pérdida de rendimiento. El uso de instrucciones SSE se deja solo como “aceptable” en caso de que estas provengan del uso de librerías *legacy*, las cuales no puedan ser reemplazadas por librerías equivalentes que hagan uso de los conjuntos AVX más modernos.

En la figura 2.9 se presenta de forma gráfica la diferencia entre los registros SIMD del KNL (SSE, AVX*x* y AVX-512) con los respectivos datos en punto flotante que caben en ellos.

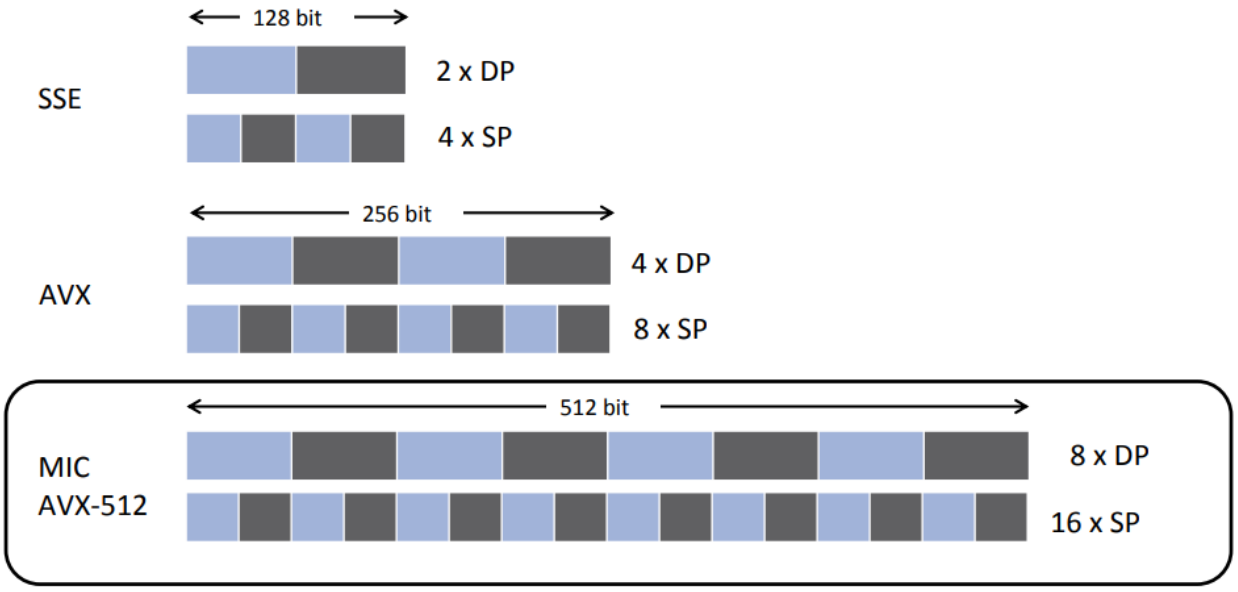

Fig 2.9: Comparación gráfica del ancho de los registros SIMD del KNL, con la cantidad y tamaño de los datos que caben en ellos.

Fuente: [28, p. 6]

Vectorización de algoritmos

Para vectorizar algoritmos para el KNL se pueden utilizar las mismas técnicas disponibles para cualquier *multi-core*:

1. Vectorización automática: también llamada “autovectorización”. Es la vectorización que realizan automáticamente los compiladores cuando están compilando en modos de optimización elevados. Se los puede ayudar mediante *flags* específicos que modifican las heurísticas del compilador, las cuales le permiten determinar en las distintas partes del código cuando sí amerita vectorizar y cuando no.
2. Uso de librerías ya vectorizadas: el uso de estas librerías brinda una forma fácil de lograr un código máquina vectorizado. No obstante, estas solo garantizan que la implementación de las funciones de las librerías estarán vectorizadas, pero no el código desde donde se las llama. Otro factor negativo, es que no siempre hay disponibles versiones vectorizadas de librerías, por lo que en muchos casos esto ni siquiera es una opción.
3. Vectorización guiada (por directivas): Mediante el uso de directivas se puede indicar al compilador específicamente que debe vectorizar un *loop* determinado. A diferencia de la vectorización automática, la vectorización guiada evita que el compilador omita la vectorización del *loop* aún si sus heurísticas detectan falsas dependencias. La principal desventaja es que queda a cargo del programador asegurarse de que no haya dependencias de datos entre iteraciones del *loop*. Si el programador fuerza al compilador a vectorizar un *loop* que efectivamente tiene dependencias entre iteraciones esto haría que el compilador entregue como resultado un programa incorrecto.
4. Vectorización manual: Es el método más difícil y complejo de vectorizar algoritmos, pero a la vez es en el que el programador tiene el mayor control sobre dónde y cómo se vectoriza. En la vectorización manual se utilizan funciones intrínsecas que al compilarlas se mapean directamente con una o un grupo de instrucciones SIMD de código máquina. Como contrapartida, si se utiliza este método se pierde portabilidad, ya que habitualmente demanda fuertes modificaciones en el código para llevar la solución a otra plataforma.

En la tabla 2.3 se pueden apreciar de forma muy sintética las diferencias básicas entre las cuatro técnicas de vectorización.

	V. Automática	V. con Librerías	V. Guiada	V. Manual
Implica modificar el código fuente	No	Sí (muy poco)	Sí (muy poco)	Sí
Facilidad de uso	Muy fácil	Fácil	Fácil	Muy difícil
Permite controlar dónde se vectoriza	Muy limitado	Solo en llamados a func. de las lib.	Sí	Sí
Permite controlar cómo se implementa la vect.	No	No	No	Sí
***[1]Libre de errores**	Sí	Sí	No	No
Es portable	Sí	Sí	Sí	No

Tabla 2.3: Tabla comparativa de técnicas de vectorización

2.3.7. Alineamiento de datos en memoria

El alineamiento de datos en memoria no es una cuestión solo a tener en cuenta como una particularidad del procesador Xeon Phi KNL, más bien siempre fue un factor relevante desde el punto de vista del rendimiento, para todo el universo de procesadores. Si se desarrolla un programa en el cual se le da importancia a su tiempo de ejecución, el alineamiento de datos en memoria es una cuestión universal a tener en cuenta.

Almacenamiento alineado

Cuando los datos se almacenan en memoria principal, estos no siempre quedan ubicados de forma alineada con respecto al comienzo de cada linea de caché. Cuando esto ocurre, podrían quedar datos partidos entre dos lineas de caché diferentes, lo que implicaría dos accesos a memoria en lugar de uno solo ante una eventual lectura/escritura de un dato; y lo que es aún peor, un dato no alineado también podría quedar separado entre dos páginas de memoria distintas.

Los compiladores normalmente solucionan el problema de alineamiento por si solos, principalmente con las estructuras de datos estáticas, agregando el *padding* necesario entre los datos hasta lograr una alineación adecuada. No obstante, en el caso de arreglos en memoria dinámica, la solución de este problema no siempre está al alcance del compilador. Hay lenguajes de programación que brindan absoluta libertad al programador en el manejo de la memoria, al permitirle utilizar los arreglos en memoria dinámica como si se tratase de un arreglo de bytes, directamente mapeado a memoria. Esta libertad deja sin la posibilidad de que en tiempo de compilación se asegure una correcta alineación de los datos. Para estos casos, los lenguajes y compiladores proveen funciones alternativas de asignación de espacio en memoria dinámica, en

1 Con “libre de errores” se hace referencia a que si bien el programador en el peor de los casos puede no lograr que el código quede vectorizado, sí se garantiza que si el código resultante efectivamente quedó vectorizado, entonces es correcto (sin errores lógicos). Desde ya que esto no contempla el caso en que el compilador o las librerías sean defectuosas, de todos modos dichos escenarios están fuera del alcance del programador.

las cuales se pasa por parámetro la cantidad de bytes con la que se quiere efectuar el alineamiento (deberá coincidir con el ancho de una linea de caché de la L1D). De esta forma, el primer elemento del arreglo quedará perfectamente alineado en la linea de caché. Luego queda a cargo del programador que los demás elementos del arreglo mantengan la alineación, pero ya la mayor parte del trabajo estará hecha; si el primer elemento está alineado, entonces el resto también lo estará si la longitud de cada uno de ellos es un divisor del ancho de la linea de caché.

Accesos alineados

Cabe destacar que no todos los procesadores soportan accesos no alineados a memoria, por lo que algunas arquitecturas tienen como requerimiento esencial que por software se asegure una alineación correcta de los datos. En el caso de los procesadores x86, éstos sí soportan lecturas no alineadas de memoria, con el objetivo de brindarle al software cierta flexibilidad en el manejo de la memoria. No obstante, esto implica una penalidad en el rendimiento, ya que además de existir la posibilidad de tener que leer frecuentemente dos líneas de caché para obtener un único dato, también le implica al procesador trabajo extra por cada operación en memoria, aún en los casos en que todos los datos se encuentran perfectamente alineados. Esto se debe a que cuando el procesador realiza accesos a memoria, a priori “no sabe” si los datos en cuestión están o no alineados, por lo que para saberlo, debe realizar ciertos chequeos, que por supuesto le implican trabajo extra. No obstante, si el procesador sabe de antemano que los datos que lee/escribe en memoria están alineados, podría ahorrarse cómputo al permitirse omitir dichas validaciones asociadas al alineamiento de datos. Esto se realiza con accesos alineados a memoria, los cuales le permiten al procesador asumir de antemano que los datos con los que está tratando están efectivamente alineados, sin necesidad de ninguna comprobación al respecto.

No obstante, el uso de accesos alineados a memoria tiene un efecto colateral; al implementar esta técnica, el procesador debe “confiar” en que los datos están efectivamente alineados, pero estos podrían no estarlo si el programador no toma los recaudos necesarios. Cuando los datos no están alineados mientras se le indica al procesador que sí lo están, podrían ocurrir errores de lectura/escritura en memoria. Por lo tanto, hay que tener especial cuidado en no hacer un mal uso de esta técnica.

En la tabla 2.4 se detallan las características de las combinaciones entre el guardado de datos de forma alineada y no alineada, con los respectivos accesos a memoria alineados y genéricos.

	Acceso normal	**Acceso alineado (asume alineamiento)**
Datos se guardan sin alineamiento	- Es la estrategia más general. - Funciona con cualquier estructura de datos sin más modificaciones. - Menor rendimiento.	- Mal uso de la técnica. - Pueden ocurrir errores de lectura/escritura en memoria.
Datos se guardan alineados	- Se asegura que cada dato a leer/escribir involucre un acceso a única linea de caché (mejor rendimiento). - Los procesadores continúan efectuando chequeos en tiempo real para saber si los datos con los que se está operando están o no alineados.	- Se asegura que cada dato a leer/escribir involucre un acceso a una única linea de caché (mejor rendimiento). - El procesador asume que los datos en memoria están alineados, por lo que puede ahorrar trabajo al omitir pasos de comprobación de alineamiento, aumentando así aún más el rendimiento.

Tabla 2.4: Estrategias de alineamiento de datos en memoria

2.3.8. Desenrollado de bucles

Al igual que el alineamiento de datos en memoria, el desenrollado de bucles es una técnica genérica para mejorar el rendimiento de algoritmos. En este caso, se trata de mejorar el tiempo de ejecución optimizando la ejecución de bucles de un programa.

Estrategia

El desenrollado de bucles se basa en eliminar cómputo innecesario en la ejecución de un bucle al reducir la frecuencia en la que se ejecutan las instrucciones de control del mismo. Dichas instrucciones son las que normalmente están involucradas en la comprobación de la condición de corte y en el incremento del índice (en el caso del *for*). La idea de esta técnica es replicar el bloque de un bucle varias veces, permitiendo que dicho bucle haga el mismo trabajo útil, pero en menor cantidad de iteraciones. Al repartirse el trabajo entre menos iteraciones, se logra disminuir la frecuencia en que se ejecutan las instrucciones de control del bucle, y por lo tanto, se reduce su *overhead* asociado*[1]. Si bien en muchos bucles estas instrucciones de control tienen un impacto marginal en el tiempo de ejecución total, no ocurre lo mismo en el caso de bucles pequeños que iteran una gran cantidad de veces. En resumen, al reducir la cantidad de iteraciones de un bucle, mejora su relación instrucciones productivas /instrucciones de control, lo que tiende a mejorar el rendimiento.

La cantidad de iteraciones que colapsan en una sola determina el *factor de desenrollado* (FD). Por ejemplo, un bucle desenrollado con un FD=4, tendrá una cuarta parte de la cantidad de iteraciones totales con respecto a su equivalente sin desenrollar, pero en su lugar, cada iteración tendrá el cuádruple de trabajo útil.

1 Las instrucciones de control de un bucle generalmente involucran accesos a memoria, y saltos condicionales; es decir, son instrucciones habitualmente sensibles con respecto al impacto en el tiempo de ejecución, por lo que el overhead que producen en muchos casos es considerable.

El desenrollado de bucles trae un efecto secundario, y es que al aumentar la cantidad de instrucciones del bloque de un bucle, se está efectivamente expandiendo el tamaño del binario ejecutable. Cuanto más grande es el valor de FD de un bucle desenrollado, más grande será el binario resultante. Este aumento del código máquina involucrado en el bucle, tiene implicancias en el rendimiento con respecto a cuestiones del uso de la memoria caché de instrucciones (L1I). Un bucle con un FD muy chico hará un uso optimo de la cache L1I, pero probablemente tendrá mayor *overhead* de sus instrucciones de control (se ejecutan más frecuentemente); mientras que un bucle con un FD muy grande, prácticamente eliminaría dicho *overhead*, pero podría hacer un uso muy ineficiente de la caché L1I. Al final de cuentas, como es lo habitual en este ámbito, la última palabra la tendrá el análisis empírico; es decir, se deben ejecutar las pruebas de rendimiento necesarias en una máquina determinada, probando con los distintos valores de FD para determinar cuál es el óptimo.

Técnicas de desenrollado de bucles

- Desenrollado de bucles automático: es realizado automáticamente por los compiladores. Estos determinan mediante heurísticas en qué bucles merece implementarse el desenrollado. Esta funcionalidad puede activarse al aumentar el nivel de optimización del compilador, o bien activarse independientemente con un parámetro específico.
- Desenrollado de bucles guiado: mediante directivas, se le puede indicar al compilador que desenrolle un bucle en particular. Además, con un parámetro opcional a las directivas, se puede asignar un valor arbitrario de FD para cada bucle a desenrollar.
- Desenrollado manual: implica modificar manualmente el código del bucle. Esta técnica en general se desaconseja, debido a que puede llegar a “confundir” al compilador, haciendo que éste no implemente adecuadamente otras optimizaciones automáticas, y por consiguiente, no se consiga el rendimiento ideal.

En la tabla 2.5 se pueden apreciar de forma muy sintética las diferencias básicas entre las tres técnicas de desenrollado de bucles.

	D. automático	D. guiado (por directivas)	D. manual
Implica modificar el código fuente	No	Sí (muy poco)	Sí
Facilidad de uso	Sí	Sí	No
Permite controlar qué bucles desenrollar	Muy limitado	Sí	Sí
FD elegido automáticamente	Sí	Sí	No
Permite elegir manualmente el FD	No	Sí	Sí
Edición manual posterior del FD	- - -	Implica alterar solo el valor de un parámetro a la directiva	Implica modificar sustancialmente el código
***[1]Libre de errores**	Sí	Sí	No

Tabla 2.5: Tabla comparativa de técnicas de desenrollado de bucles

2.4. Resumen

En este capítulo se hizo una introducción al procesador Xeon Phi KNL y su arquitectura, presentando a éste como un procesador fuertemente orientado al cómputo paralelo, y capaz de ejecutar cualquier programa x86 sin modificaciones. Esta característica, además de darle la ventaja de mantener compatibilidad con software *legacy*, permite reducir el esfuerzo de programación requerido para desarrollar software optimizado para esta arquitectura. Al desarrollar una aplicación para el KNL, se pueden utilizar las mismas técnicas y herramientas de optimización utilizadas habitualmente para los *multi-cores*, así como también sus mismos modelos de programación paralela, como es el caso de OpenMP, el cual tiene toda una sección dedicada en este capítulo. Esta retrocompatibilidad, además de permitir desarrollar software "desde cero" para el KNL haciendo uso de los modelos de programación para *multi-cores* más maduros y desarrollados, también permite literalmente partir de aplicaciones ya desarrolladas y optimizadas para *multi-cores*; de modo que luego, con relativamente pocos ajustes y optimizaciones adicionales, se pueden conseguir importantes ganancias de rendimiento al ejecutar estas aplicaciones en un Xeon Phi KNL.

A la hora de desarrollar una aplicación para el KNL, lo que primero se debe elegir es el modo cluster y el modo de memoria a utilizar. Éstos definirán cómo se debe proseguir con el resto de las optimizaciones.

1 Con "libre de errores" se hace referencia a que si bien el programador en el peor de los casos puede no lograr que el bucle quede desenrollado, sí se garantiza que si el bucle quedó efectivamente desenrollado, entonces es correcto (sin errores lógicos). Desde ya que esto no contempla el caso en que el compilador o las librerías sean defectuosas, de todos modos dichos escenarios están fuera del alcance del programador.

El modo cluster a elegir estará primero determinado por la configuración de módulos DDR instalados en el sistema. Si no todos los módulos son idénticos en capacidad, solo el modo *All-to-all* estará disponible. En caso contrario, se podrán usar los otros modos clusters más rápidos. Si se trata de una aplicación MPI con 4 o 2 procesos por nodo, puede probarse el modo SNC-4 (o el SNC-2 respectivamente), ya que en dichos casos los modos SNC tienen potencial de funcionar más rápido que *Quadrant*. No obstante, si ninguna de estas condiciones se cumple, queda por descarte el modo *Quadrant* (y *Hemisphere*), el cual a fines prácticos, será el modo cluster a utilizar en la gran mayoría de los casos. Con *Hemisphere* muy rara vez se obtenga mejor rendimiento que con *Quadrant*, por lo que este modo es relegado como una variante específica de *Quadrant* mucho menos utilizada.

Por otro lado, el modo de memoria a utilizar dependerá en primer lugar de la cantidad máxima de memoria que vaya a utilizar el programa a ejecutar. Si este no demanda más de 16 GB, puede ser ejecutado en modo *flat* haciendo uso solamente de la memoria MCDRAM, aprovechando así de forma simple y directa todo su ancho de banda. Cuando el programa demanda más de 16 GB (algo muy habitual en HPC) el modo de memoria *cache* será el ideal en la mayoría de estos casos. Con el modo de memoria *cache*, el software puede gozar del ancho de banda de la MCDRAM mientras se abstrae del manejo de la misma. No obstante, en algunos casos, en aplicaciones que superan ampliamente los 16 GB de espacio requerido en memoria y que a la vez tienen una baja taza de hits (por ser de naturaleza "*cache-unfriendly")*, con el modo *flat* (o *hybrid*) probablemente puedan obtener más rendimiento. El inconveniente de estos modos de memoria *flat* y *hybrid*, es que en dichos casos en los que el programa demanda más de 16 GB, queda como tarea del programador hacer un uso discrecional de ambos tipos de memoria (DDR y MCDRAM). Esto requiere un análisis mucho más profundo para entender las características de uso de memoria del algoritmo, para así poder distribuir "manualmente" los datos (entre ambos tipos de memoria) de la forma más "amigable" posible al *working-set* del proceso. En pocas palabras, la idea con respecto al uso de memoria en el KNL es lograr que siempre quede el *working-set* en la MCDRAM, ya sea ésta manejada de forma automática (modo *cache*), de forma manual (*flat*), o bien mixta (*hybrid*).

Por último, como se vio en más detalle a lo largo de este capítulo, las optimizaciones genéricas para todo *multi-core* (así como también para todo procesador mono-núcleo) tampoco pueden faltar en una aplicación optimizada para KNL. Entre ellas se destacan la vectorización, el alineamiento de datos en memoria, la afinidad de hilos con núcleos, y el desenrollado de bucles.

Capítulo 3

Algoritmos para caminos mínimos en grafos

En este capítulo, primero se enuncia el problema de caminos mínimos en grafos (sección 3.1). Luego se presentan y describen brevemente distintos algoritmos que resuelven dicho problema (sección 3.2). Posteriormente se presenta el estado del arte de la aceleración del algoritmo FW (sección 3.3), y por último, se presenta un resumen del capítulo (sección 3.4).

3.1. Problema de caminos mínimos en grafos

El problema a tratar en este trabajo es el "*problema del camino mínimo*", "*problema del camino más corto*", o como se lo llama en inglés "*shortest path problem*" de teoría de grafos, pero aplicado a todos los vértices del grafo de entrada. Es decir, se debe hallar el camino mínimo entre todos los pares de vértices del grafo. A esta clase de problemas se la conoce como "*All pairs shortest path*" (APSP), que se lo puede traducir como "*camino mínimo entre todos los pares* (de vértices)". Por otro lado, si se deseara sólo hallar el camino mínimo a partir de un único vértice, en ese caso se trata de la clase de problemas "*Single Source Shortest Path*" (SSSP), que se lo puede traducir como "*Camino mínimo a partir de un único origen*".

Encontrar los caminos más cortos es un problema importante y fundamental en comunicaciones [6], redes de transporte [5], diseño de circuitos [9], redes sociales [10], entre otros, y es un subproblema de muchos problemas combinacionales, los cuales pueden ser representados como un problema de flujos de red.

Las computaciones implicadas en el cálculo de los caminos mínimos en las diversas aplicaciones que lo utilizan, involucran una porción grande del tiempo de ejecución total de la aplicación. Queda claro que el cálculo de caminos mínimos es un evidente *hotspot**[1] en las distintas aplicaciones, por lo que merece un análisis adecuado para desarrollar una solución paralela y optimizada (sobretodo en el ámbito de HPC), a modo de reducir sustancialmente el tiempo de ejecución de dichas aplicaciones.

El problema del camino mínimo de un grafo consiste en encontrar el camino más corto entre dos de sus vértices; es decir, se necesita que la suma de los pesos de las aristas que componen el camino sea la mínima. Por ejemplo, encontrar el camino más corto entre dos ciudades (las ciudades se representarían como vértices, las rutas como aristas, y el peso de dichas aristas estaría representado por ejemplo en kilómetros).

1 *Hotspot*: parte pequeña de un programa que se ejecuta una gran cantidad de veces. Los hotspots son los principales objetivos donde deben focalizarse las tareas de optimización.

3.2. Algoritmos para caminos mínimos en grafos

A continuación se presentan algunos de los algoritmos más comunes de caminos mínimos de grafos.

Notación:

|E| = Cantidad de aristas del grafo
|V| = Cantidad de vértices del grafo

Algoritmos SSSP

- BFS (Breadth First Search): Se lo puede utilizar para hallar el camino mínimo en grafos no pesados. Tiene un tiempo de ejecución de O(|E|+|V|).
- Dijkstra: Algoritmo de programación dinámica*[1]. Funciona con grafos pesados, pero sólo admite pesos positivos. El tiempo de ejecución de una de sus versiones más optimizadas es de O(|E| + |V|*log(|V|)).
- Bellman-Ford: Algoritmo de programación dinámica. Funciona con grafos pesados y admite pesos negativos. Tiene un tiempo de ejecución de O(|E|*|V|).

Algoritmos APSP

- Johnson: Reutiliza los algoritmos Dijkstra y Bellman-Ford para resolver APSP. Ejecuta primero una pasada de un Bellman-Ford modificado con el cual reescribe el peso de las aristas; de modo que luego, al ejecutar |V| veces el algoritmo Dijktra (una vez por cada vértice), éste pueda funcionar correctamente, ya que no estarían presentes los pesos negativos originales. Este algoritmo resuelve el problema APSP con un tiempo de ejecución de $O(|E|*|V| + |V|^2*\log(|V|))$.
- Floyd-Warshall: Algoritmo de programación dinámica. Funciona con grafos pesados y admite pesos negativos. Tiene un tiempo de ejecución de $O(|V|^3)$.

Cabe destacar que los tiempos de ejecución dependientes de |E| mencionados anteriormente, como no dependen únicamente de la cantidad de vértices sino también de la cantidad de aristas, el tiempo de ejecución puede variar sensiblemente entre un grafo denso y uno disperso. En efecto, un grafo denso tiende a tener $|E| = |V|^2$, por lo que en dichos casos los tiempos de ejecución se vuelven al menos de $O(n^3)$ (n^2 en el caso de Dijkstra) con respecto a la cantidad de vértices.

1 Programación dinámica: es una técnica matemática orientada a la solución de problemas en los cuales se toma decisiones de forma secuencial. Esta técnica se centra en reducir sustancialmente el costo de la solución (complejidad espacial y/o temporal). A fines prácticos, para lograrlo, los algoritmos de programación dinámica a menudo guardan resultados intermedios en memoria, para que estos sean reutilizados en etapas futuras de la ejecución, sin que tengan que volver a calcularse como si se tratasen de subproblemas totalmente diferentes. No obstante, no todo problema puede resolverse con programación dinámica; para esto debe cumplirse el principio de optimalidad de Bellman [11].

Johnson vs Floyd-Warshall

Si bien para resolver el problema APSP hay otras alternativas, en la mayoría de los casos se suele optar por una de estas dos soluciones. Lo que determina principalmente cuál algoritmo será el más apropiado para un dominio particular, es el costo computacional; es decir, normalmente será elegido el que tiene menor tiempo de ejecución según las características de los parámetros de entrada para ese dominio. La respuesta es sencilla: para grafos dispersos, suele ser mejor el algoritmo de Johnson, mientras que para grafos densos, el algoritmo Floyd-Warshall en general es el más eficiente.

Como naturalmente los grafos densos demandan más poder de cómputo que los dispersos, para este trabajo resulta más interesante centrarse en estudiar el algoritmo Floyd-Warshall, ya que como se dijo en el párrafo anterior, de base se trata del más eficiente para estos casos. Luego, a lo largo de este trabajo, se desarrollará una paralelización del algoritmo, implementando además sucesivas optimizaciones a modo de buscar el mejor rendimiento posible a la hora de ser ejecutado en un Xeon Phi KNL.

3.2.1. Algoritmo Floyd-Warshall

El algoritmo FW es un algoritmo de programación dinámica que resuelve el problema APSP descripto anteriormente. Este algoritmo encuentra el camino mínimo entre todos los pares de vértices de un grafo dirigido (o un digrafo) en una única ejecución. En términos simples, el algoritmo FW obtiene la matriz de caminos mínimos resultante en $O(n^3)$ computaciones, y utilizando un espacio de memoria de $O(n^2)$.

Como se puede ver en el pseudocódigo mostrado en la figura 3.1, el algoritmo FW “clásico” consiste en solo tres bucles anidados con un *IF* en el bucle más interno. Toma como entrada a la matriz de distancias mínimas (D), la matriz de reconstrucción de caminos mínimos (P), y el número entero N que representa a la cantidad de vértices del grafo.

Pseudocódigo

```
1:  for k = 0..N-1 do
2:      for i = 0..N-1 do
3:          for j = 0..N-1 do
4:              if D_{i,k} + D_{k,j} < D_{i,j} then
5:                  D_{i,j} = D_{i,k} + D_{k,j}
6:                  P_{i,j} = k
7:              end if
8:          end for
9:      end for
10: end for
```

Fig 3.1: Pseudocódigo del algoritmo FW secuencial "clásico".

Fuente: Fig 1 [27, p. 4]

Detalle:

- D: matriz de distancias mínimas.
- P: matriz de reconstrucción de caminos mínimos.
- i: índice del vértice origen de un camino a analizar.
- j: índice del vértice destino de un camino a analizar.
- k: índice de un vértice intermedio del camino a analizar entre los vértices i y j.
- N: cantidad de vértices del grafo.

Descripción de la sentencia *IF*:

Si la longitud del camino entre el vértice origen (i) y destino (j) definida en la matriz de entrada es mayor que la longitud del camino alternativo pasando por el vértice intermedio (k); entonces se actualiza la distancia mínima con la longitud de éste camino alternativo más corto. Además, también se guarda el índice del vértice k en la matriz P para poder luego eventualmente reconstruir el camino mínimo en caso de ser necesario.

Luego de ejecutar la sentencia *IF* para todo i, todo j y todo k, se garantiza que en la matriz D (ahora modificada) se obtiene la distancia mínima entre cada par de vértices, mientras que los respectivos caminos mínimos se obtienen en la matriz P.

3.3. Estado del arte sobre aceleración del algoritmo Floyd-Warshall

La aceleración de FW es un problema ampliamente estudiado por la comunidad científica. Podemos encontrar soluciones sobre diferentes arquitecturas. A continuación, se describen las más destacadas, haciendo hincapié en las que emplean aceleradores Xeon Phi.

Las primeras implementaciones aceleradas de FW estuvieron orientadas a arquitecturas monoprocesador. Tanto Venkataraman et al. [12], como Park et al. [13] mostraron que, bajo ciertas condiciones, es posible reordenar el cómputo de las celdas para implementar técnicas de *blocking*. Este reordenamiento permite una mayor explotación de la localidad de datos, lo que aumentó el rendimiento aproximadamente 2x. En sentido similar, Han and Kang [14] y Han et al. [15] demostraron que, el uso de instrucciones vectoriales (en particular de la familia SSE) en combinación con técnicas de desenrollado de bucle, pueden mejorar el rendimiento hasta 5.7x.

Más adelante en el tiempo, se pueden encontrar implementaciones para arquitecturas multiprocesador de memoria compartida, de memoria distribuida y de memoria híbrida. En [16] se propone una versión paralela de FW usando el estándar MPI. Los autores obtienen aceleraciones de 2.8x y de 1.2x al emplear 16 y 32 nodos del cluster. En [17] se paraleliza el algoritmo FW empleando la librería TBB sobre un arquitectura dual-core. Posteriormente, con la incorporación de procesadores *multi-core* a los clusters, también se exploró su uso para acelerar FW mediante la combinación de MPI con OpenMP [18].

Uno de los primeros trabajos en evaluar el uso de FPGAs (Field Programmable Gate Arrays) para computar FW es Bondhugula et al. [19], donde se implementa una versión por bloques que

logra una mejora de 22x sobre un procesador Cray XD1. En Chen et al. [20] se presenta una solución híbrida CPU+FPGA, que permite aprovechar lo mejor de ambas arquitecturas.

Las GPUs son el acelerador dominante en la actualidad y varios trabajos investigaron cómo aprovecharlas para computar FW. En [21], Harish y Narayan presentaron una implementación CUDA sencilla para el algoritmo FW clásico (sin aplicar *blocking*). Posteriormente, podemos destacar la propuesta de Katz y Kider [22], quienes presentaron una implementación eficiente por bloques que permite manejar grafos que requieren más memoria que la disponible en el dispositivo. En [23], Ben y Lund intensifican el uso de los diferentes niveles de memoria de las GPUs para así mejorar las soluciones existentes al momento. Posteriormente, con la aparición de arquitecturas híbridas CPU-GPU, surgieron nuevas implementaciones, como la de Matsumoto et al. [24] que estudió cómo minimizar la comunicación *host-device*. Recientemente, algunos trabajos han explorado soluciones con múltiples GPUs, como el de Djidjeva et al [25].

Por último, en arquitecturas Xeon Phi se pueden mencionar 2 propuestas. Hou et al. [26] propuso una implementación OpenMP para coprocesadores KNC. Los autores encontraron que el uso de *blocking* y vectorización son fundamentales para obtener alto rendimiento. También que la vectorización guiada da mejores resultados que la manual. Rucci et al. [27] exploró el uso de procesadores KNL para acelerar FW. Al igual que el trabajo anterior, los autores utilizaron OpenMP para paralelizar la solución. También en sentido similar, destacan que explotar localidad de datos mediante *blocking* y el uso de instrucciones vectoriales resultan factores fundamentales para mejorar el rendimiento. Finalmente, remarcan la importancia del uso de la memoria MCDRAM para tolerar las demandas de memoria de alto ancho de banda.

3.4. Resumen

En este capítulo se presentó el problema de hallar los caminos mínimos en grafos, detallando las respectivas clases de problemas APSP y SSSP. Luego se hizo un relevamiento de los distintos algoritmos pertenecientes a dichas clases, analizando las principales características y tiempos de ejecución. Dentro de los algoritmos APSP, se hizo énfasis en la comparación entre FW y el algoritmo de Johnson. Como se analizó en este capítulo, cuál de estos dos algoritmos resulta más apropiado para resolver el problema APSP dependerá de las características del grafo de entrada. Para grafos dispersos, puede funcionar más rápido el de Johnson, mientras que para grafos densos, FW es más eficiente. Por lo tanto, desde el punto de vista del tiempo de ejecución en el peor caso, el algoritmo FW es el mejor, característica por la cual finalmente será elegido como el algoritmo a desarrollar en este trabajo.

Luego en este capítulo se dio una introducción al estado del arte de la aceleración del algoritmo FW, dando un panorama de las implementaciones disponibles para distintas plataformas habituales de HPC, tales como las GPUs, *multi-cores*, FPGAs y los Xeon Phi. Las soluciones para *multi-cores* y Xeon Phi son las que serán tenidas en cuenta a la hora de investigar y tomar de referencia para este trabajo, ya que las GPUs y FPGAs presentan diferencias radicales de hardware, que se traducen en modelos de programación esencialmente diferentes al del resto de los procesadores.

Capítulo 4

Aceleración del algoritmo Floyd-Warshall sobre Xeon Phi KNL

En este capítulo, primero se presenta la plataforma de pruebas donde se corrieron los experimentos (sección 4.1). Luego se listan las distintas versiones de dichos experimentos, con el conjunto de valores de prueba que se utilizaron para cada uno de los parámetros en las distintas ejecuciones (sección 4.2). Prosigue la presentación de la implementación y los resultados de rendimiento de las diferentes versiones detalladas en la sección 4.2 con su análisis correspondiente (secciones 4.3, 4.4 y 4.5). Luego se comparan los resultados con los obtenidos en otros trabajos similares (sección 4.6); y por último, se presenta un resumen del capítulo (sección 4.7).

4.1. Plataforma de pruebas

Las pruebas se ejecutaron en un Intel Xeon Phi 7230 configurado en modo cluster *All-to-All*, modo de memoria *flat*, y con una configuración de 128 GB de RAM DDR. El sistema operativo utilizado fue Ubuntu 16.04.3, y el código de los experimentos se compiló con el compilador ICC de Intel (versión 19.0.0.117). En la implementación de las versiones paralelas se utilizó OpenMP como herramienta principal de paralelización.

4.2. Configuraciones de las pruebas

A continuación se detallan las distintas versiones del algoritmo utilizadas para las pruebas de rendimiento. El desarrollo se llevó a cabo partiendo del algoritmo Floyd-Warshall “clásico” en su versión secuencial y paralela como primera referencia; luego, se desarrolló una variante del mismo que hace uso de técnicas de *blocking*, también en su versión secuencial y paralela. Una vez desarrollado el algoritmo paralelo con *blocking*, se lo toma a éste como base para un desarrollo iterativo e incremental de distintas versiones optimizadas. En el desarrollo de cada versión se analizan y comparan empíricamente el rendimiento y tasa de mejora de cada una de ellas. Todas estas versiones aquí mencionadas se presentan en las tablas 4.1 y 4.2.

Más adelante en este trabajo, se muestran los resultados de experimentos adicionales que prueban distintos parámetros y configuraciones a los utilizados en las versiones *Opt-X*. Estos experimentos adicionales (se presentan en la tabla 4.3) si bien están basados en la versión *Opt-8*, son independientes entre sí; es decir, no son optimizaciones incrementales. Por ejemplo, el experimento *Var-Types* no reutiliza ninguna modificación realizada por *Var-Prec* ni viceversa, pero sí ambos parten del código de la versión *Opt-8*.

Versión	Descripción
Naive-Sec	Floyd-Warshall “clásico”
Naive-Par	Paralelización simple del Floyd-Warshall “clásico”
Block-Sec	Variante de Floyd-Warshall que utiliza *blocking* (secuencial)

Tabla 4.1: Primeras versiones de FW implementadas

Versión	Descripción
Opt-0	Floyd-Warshall con *blocking* (paralelo)
Opt-1	Optimización haciendo uso de la memoria MCDRAM
Opt-2	Optimización utilizando vectorización guiada (SSE)
Opt-3	Optimización utilizando vectorización guiada (AVX2)
Opt-4	Optimización utilizando vectorización guiada (AVX512)
Opt-5	Optimización utilizando alineación de datos en memoria
Opt-6	Optimización utilizando predicción de saltos por software
Opt-7	Optimización utilizando desenrollado de bucles
Opt-8	Optimización ajustando la afinidad de hilos con núcleos

Tabla 4.2: Version Opt-0 y sus optimizaciones incrementales

Versión	Descripción
Var-NoPath	Se prueba omitiendo la matriz de reconstrucción de caminos mínimos (es decir, sólo se calcula la matriz de distancias)
Var-Dens	Se prueba con diferentes grados de densidad del grafo de entrada
Var-Prec	Se prueba relajando la precisión de *float*
Var-Double	Se prueba con el tipo de dato *double* (para el peso de las aristas)

Tabla 4.3: Experimentos adicionales

Las versiones secuenciales *Naive-Sec* y *Bloq-Sec* se desarrollaron sólo para tenerlas de referencia, pero no se efectuó ningún análisis empírico con las mismas. Normalmente no tiene mucho sentido ejecutar programas mono-hilo en un procesador fuertemente orientado al cómputo paralelo, como es éste el caso con el Xeon Phi KNL; por lo que sólo se corrieron pruebas y se compararon resultados de las versiones paralelas.

La versión *Naive-Par* es la primera a ejecutar y a tomar sus resultados de tiempo y GFLOPS*[1] como primera referencia.

La versión *Opt-0* es la que servirá de base para las demás. El resto de las versiones *Opt-X* (*Opt-1, Opt-2,* etc) son optimizaciones incrementales de la versión *Opt-0;* es decir, la versión *Opt-2*

1 GFLOPS = Mil millones de operaciones de punto flotante por segundo.

contiene todas las optimizaciones implementadas en la versión *Opt-1* mientras agrega optimizaciones nuevas; la versión *Opt-3* contiene todas las optimizaciones de las versiones *Opt-1*, *Opt-2* mientras agrega nuevas, y así sucesivamente.

Exceptuando el experimento *Var-Nopath*, en todos los casos no solo se computará la matriz de distancias (matriz ***D***), sino que también se calculará la matriz de caminos mínimos (matriz ***P***).

Valores de N de prueba: **4096**, **8192**, **16384**, **32768** y **65536**.

Valores de T (cantidad de hilos) de prueba: **64**, **128** y **256**; para utilizar SMT con 1, 2 y 4 hilos por núcleo respectivamente.

No se utilizará el T de prueba T=192 ya que, como se especificó en la sección 2.3.3, la configuración de 3 hilos por núcleo muy rara vez resulta óptima. Además, como en este caso los valores de N y BS*[1] serán potencias de 2, es mejor que los valores de T también lo sean, para que la repartición de trabajo entre hilos sea óptima independientemente del valor de T a utilizar en las pruebas.

Valores de BS de prueba*[2]: **32**, **64**, **128** y **256**.

Porcentaje de densidad del grafo: exceptuando el experimento *Var-Dens*, en todos los casos se utiliza una densidad del 70%.

Tipo de dato utilizado para el peso de las aristas: exceptuando el experimento *Var-Types*, en todos los casos se utiliza el tipo de dato *float*.

Cantidad de veces que se ejecuta una prueba en una configuración determinada: 5.

Cómo se calculan los resultados: se toman los promedios del tiempo de ejecución y de los GFLOPS de las sucesivas ejecuciones de cada configuración que se esté probando (dentro de una misma versión del programa). Los resultados arrojados con la configuración que resulte óptima serán establecidos como los resultados de rendimiento finales de la versión.

Cómo se muestran los resultados: En las distintas versiones y experimentos se mostrarán los resultados en GFLOPS con sus respectivos gráficos que resulten adecuados según el caso. Además, para cada versión probada se mostrará la tasa de mejora obtenida con respecto a su anterior versión con menos optimizaciones.

En algunos casos, puede que para una misma versión una configuración que resulta óptima con un valor de N determinado no resulte ser la mejor para otros valores de N. No obstante, por simplicidad y trazabilidad a lo largo de las diferentes versiones, se eligirá una única combinación de parámetros optima por versión, independientemente de N. Para llevar a cabo esta elección, siempre se le dará más peso al resultado arrojado con los valores de N más grandes, ya que como es lo habitual en HPC, se prioriza agilizar el cómputo de algoritmos para tamaños de entrada grandes.

1 BS = “Tamaño de bloque”. Es un parámetro que se utiliza a partir de la versión “Block-Sec”.
2 El por qué de éstos valores de BS de prueba arbitrarios se explica más adelante en la sección 4.4

4.3. Primeras versiones de FW implementadas

4.3.1. Versión *Naive-Sec*: FW Secuencial “clásico”

Implementación

```
1  void floydWarshall(TYPE* D, int* P, int n){
2      INT64 i, j, k, i;
3      TYPE sum;
4
5      for(k=0; k<n; k++){
6          for(i=0; i<n; i++){
7              for(j=0; j<n; j++){
8                  sum = D[i][k] + D[k][j];
9                  if(sum < D[i][j]){
10                     D[i][j] = sum;
11                     P[i][j] = k;
12                 }
13             }
14         }
15     }
16 }
```

Fig 4.1: Captura de implementación de FW secuencial "clásico"

El Floyd-Warshall “clásico” desarrollado en este trabajo (figura 4.1), es la implementación en C del pseudocódigo de FW visto en la figura 3.1 del capítulo anterior. Por más que esta versión clásica del algoritmo no sea la más rápida, de todos modos sigue siendo muy utilizada gracias a su facilidad de implementación; por lo que si los grafos de entrada no son grandes, ésta implementación muy a menudo satisface las necesidades. No obstante, en el ámbito de HPC, los tamaños de entrada muy grandes son característicos; por lo que si se necesita implementar una versión de FW en una aplicación de HPC, se debe empezar a pensar en realizar fuertes optimizaciones, o directamente emplear versiones alternativas del algoritmo, como se verá más adelante.

Organización de las matrices en memoria

En los *hotspots* de un programa siempre es deseable acceder a las estructuras de datos de la forma más eficiente posible; sin embargo, el rendimiento real dependerá fuertemente de cómo dichas estructuras son realmente guardadas en la memoria principal. Por lo tanto, cuanto a más “bajo nivel” el programador maneje dichas estructuras, más herramientas tendrá para lograr que su programa haga un uso óptimo de la memoria.

En el caso de las matrices, si se guardan en vectores simples en lugar de arreglos de dos dimensiones, es posible lograr así un mayor control sobre cómo se accede realmente a los datos en memoria.

A modo de ejemplo, en la tabla 4.4 se presenta una matriz de 4x4. Luego, en la tabla 4.5 se muestra a la misma matriz, pero esta vez representada en forma de vector de una sola dimensión.

[Fila, Columna]	Columna 0	Columna 1	Columna 2	Columna 3
Fila 0	[0,0]	[0,1]	[0,2]	[0,3]
Fila 1	[1,0]	[1,1]	[1,2]	[1,3]
Fila 2	[2,0]	[2,1]	[2,2]	[2,3]
Fila 3	[3,0]	[3,1]	[3,2]	[3,3]

Tabla 4.4: Ejemplo de matriz cuadrada de N=4

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
[0,0]	[0,1]	[0,2]	[0,3]	[1,0]	[1,1]	[1,2]	[1,3]	[2,0]	[2,1]	[2,2]	[2,3]	[3,0]	[3,1]	[3,2]	[3,3]

Tabla 4.5: Matriz cuadrada de N=4 guardada en un vector de forma ordenada por filas

```
1 void floydWarshall(TYPE* D, int* P, int n){
2     INT64 i, j, k, i;
3     TYPE sum;
4
5     for(k=0; k<n; k++){
6         for(i=0; i<n; i++){
7             for(j=0; j<n; j++){
8                 sum = D[i*n+k] + D[k*n+j];
9                 if(sum < D[i*n+j]){
10                    D[i*n+j] = sum;
11                    P[i*n+j] = k;
12                }
13            }
14        }
15    }
16 }
```

Fig 4.2: Captura de pantalla de FW "clásico" con vectores unidimensionales

Partiendo de la implementación en C más básica que se desarrolló anteriormente (figura 4.1), ahora se reemplazan los arreglos bidimensionales por vectores de una sola dimensión. La implementación resultante quedaría como se observa en el código de la figura 4.2. Al utilizar los vectores 1D, el programador tiene más control sobre cómo se guardan los datos en memoria y cómo se calculan los índices, lo que le permite implementar optimizaciones más fácilmente.

Ahora que ya se tiene un manejo manual de los índices, se puede proceder a eliminar cómputo redundante.

```
1  void floydWarshall(TYPE* D, int* P, int n){
2      INT64 i, j, k, i_disp, k_disp;
3      TYPE dij, dik, dkj, sum;
4
5      for(k=0; k<n; k++){
6          k_disp = k*n;
7          for(i=0; i<n; i++){
8              i_disp = i*n;
9              dik = D[i_disp+k];
10             for(j=0; j<n; j++){
11                 dij = D[i_disp+j];
12                 dkj = D[k_disp+j];
13                 sum = dik + dkj;
14                 if(sum < dij){
15                     D[i_disp+j] = sum;
16                     P[i_disp+j] = k;
17                 }
18             }
19         }
20     }
21 }
```

Fig 4.3: Captura de pantalla de FW "clásico" optimizado ahorrando cómputo redundante (versión Naive-Sec)

Como se puede apreciar en el código de la figura 4.3, efectivamente se ahorra cómputo en el cálculo de los índices dentro de los *loops*. Por ejemplo, la multiplicación k*n ya no se calcula n^3 veces, ahora se la calcula solo cuando es necesario, es decir, n veces (ya que está en el *loop* más externo), guardando el resultado en *k_disp* para que luego éste sea reutilizado en los *loops* más internos. Lo mismo sucede con *i_disp*, que ejecuta n^2 multiplicaciones, cuando antes de esta optimización se ejecutaba n^3 veces.

4.3.2. Versión *Naive-Par*: FW “clásico” paralelizado

Ésta solución paralela es la resultante de paralelizar el algoritmo FW secuencial “clásico”.

Implementación

Tomando de base al FW secuencial “clásico”, se optó por paralelizar el segundo *loop* (el que itera entre filas), como se ve en la captura 4.4. También se podría haber paralelizado el tercer *loop* si se quisiera una variante de grano más fino. No obstante, no es posible paralelizar el *loop* más externo (el que itera sobre k), ya que hay dependencias entre cada iteración del mismo.

```
1 void floydWarshall(TYPE* D, int* P, int n, int t){
2     #pragma omp parallel default(none) firstprivate(D,P,n) num_threads(t)
3     {
4         INT64 i, j, k, i_disp, k_disp;
5         TYPE dij, dik, dkj, sum;
6         for(k=0; k<n; k++){
7             k_disp = k*n;
8             #pragma omp for schedule(dynamic)
9             for(i=0; i<n; i++){
10                i_disp = i*n;
11                dik = D[i_disp+k];
12                for(j=0; j<n; j++){
13                    dij = D[i_disp+j];
14                    dkj = D[k_disp+j];
15                    sum = dik + dkj;
16                    if(sum < dij){
17                        D[i_disp+j] = sum;
18                        P[i_disp+j] = k;
19                    }
20                }
21            }
22        }
23    }
24 }
```

Fig 4.4: Captura de pantalla de paralelización simple del FW secuencial "clásico" (versión Naive-Par)

Reparto de trabajo entre hilos

Al ser la planificación estática (es la modalidad por defecto de OpenMP), las filas se distribuyen entre los hilos en partes iguales. La cantidad de filas a cargo de cada hilo sería la siguiente: filas_por_hilo = ***N***/***T***, siendo **N** la cantidad de vértices del grafo, y ***T*** la cantidad de hilos.

Representación gráfica de la distribución de trabajo entre los diferentes hilos (para 4 hilos):

Como se detalló anteriormente, la matriz en realidad se guarda en un vector para gozar de mayor control sobre cómo se acceden a los datos, a modo de lograr un acceso a memoria secuencial. El gráfico de la matriz guardada en un vector quedaría de la siguiente manera:

Los colores representan exactamente lo mismo de antes, es decir, en la parte azul del vector se encuentran las filas de la matriz a cargo del hilo 1, en la parte violeta se encuentran las filas de la matriz a cargo del hilo 2, etcétera.

Configuraciones a probar

T: 64, 128 y 256.

Resultados

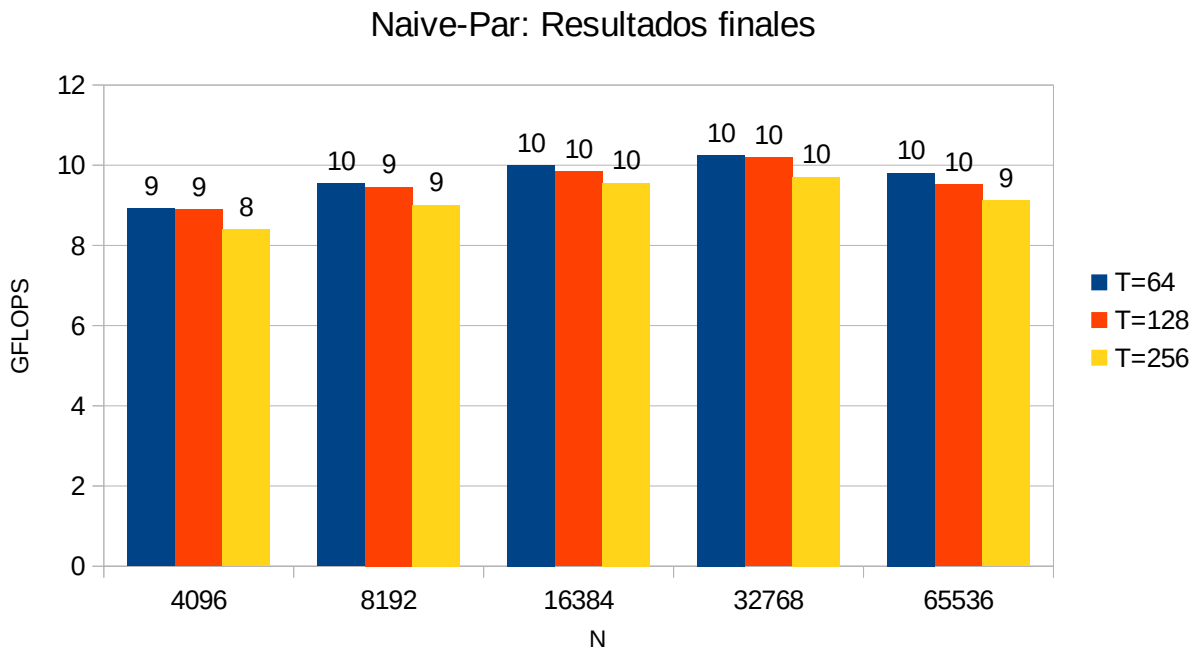

Fig 4.5: Resultados finales de la versión Naive-Par

Como se puede apreciar en el gráfico de resultados finales de la versión *Naive-Par* (Fig 4.5), el algoritmo funcionó mejor con 64 hilos que con 128 y 256. Cabe destacar que con 64 hilos se está ejecutando una configuración de 1 hilo por núcleo; es decir, no se está haciendo uso de la tecnología SMT.

Como esta versión del algoritmo tiene una localidad temporal y espacial muy mala, agregar más hilos a los mismos núcleos solo hace que se produzcan más fallos de caché L1D. Además, en dicho caso, los hilos que compartirían los mismos núcleos, tratarían de acceder todo el tiempo a posiciones muy distintas entre sí. De esta forma, se estarían “pisando” los bloques de caché todo el tiempo al resolver cada fallo de caché, por lo que no se estaría haciendo un uso colaborativo de la misma, condición esencial para conseguir ganancia de rendimiento haciendo uso de SMT.

4.3.3. Versión *Block-Sec*: Variante de FW con *blocking*

El problema que presenta el algoritmo Floyd-Warshall "clásico" es que tiene una localidad de datos deficiente, es decir, accede frecuentemente a posiciones muy distantes por toda la matriz. Para grafos grandes (N>2048), donde las matrices dejan de entrar completamente en la memoria caché L2, el FW "clásico" tiene una merma del rendimiento en este procesador, empeorando aún más a medida que N crece.

Para mejorar la localidad espacial de datos se pueden implementar técnicas de *blocking* con las matrices, lo cual en el caso de FW implica cambios bastante profundos en el algoritmo. Un algoritmo FW modificado que hace uso de técnicas de *blocking* permite elegir un tamaño de bloque acorde a la capacidad de los distintos niveles de memoria caché, independientemente del tamaño de N. De esta forma se mejora la localidad de datos, y por lo tanto se optimiza el uso de memoria caché.

Para implementar *blocking* en el algoritmo FW surgen algunos inconvenientes. El problema principal es que existen dependencias entre cada iteración del bucle que itera sobre ***k*** (los bucles de ***i*** y de ***j*** pueden ejecutarse en cualquier orden sin ningún problema); por lo que a priori, dicho bucle no puede fraccionarse en bloques de modo de poder ejecutarlos libremente de forma concurrente, ya que datos de un bloque dependerían de los resultados parciales de otros bloques. Sin embargo, como veremos a continuación, bajo ciertas condiciones, el *loop* de ***k*** puede ser insertado dentro del *loop* de ***i*** y el de ***j***, haciendo posible el *blocking*.

Algoritmo

Síntesis de Floyd-Warshall con blocking

A la matriz de distancias ***D*** se la particiona en bloques de tamaño ***BS*** **x** ***BS***, siendo ***BS*** el tamaño de bloque por lado, por lo que hay $(N/BS)^2$ bloques. Las computaciones involucran $R = N/BS$ rondas, donde cada ronda es dividida en cuatro etapas, basado en la dependencia de datos entre los bloques:

1. Actualizar el bloque ***k***,***k*** ($D^{k,k}$), ya que solo es dependiente de si mismo.
2. Actualizar los bloques restantes de la fila k-ésima, ya que cada uno de esos bloques depende de si mismo y del bloque $D^{k,k}$ computado previamente.
3. Actualizar los bloques restantes de la columna k-ésima, ya que cada uno de esos bloques depende de si mismo y del bloque $D^{k,k}$ computado previamente.
4. Actualizar el resto de los bloques de la matriz, ya que cada uno de ellos depende del k-ésimo bloque de su fila y del k-ésimo bloque de su columna, y estos ya fueron computados previamente.

Fuente: Sección 4.1 [27, p. 4]

Respetando la separación en etapas como se especificó en el cuadro anterior, se satisfacen así todas las dependencias del algoritmo. La figura 4.6 muestra de forma esquemática la computación de una ronda y las dependencias de datos entre los bloques.

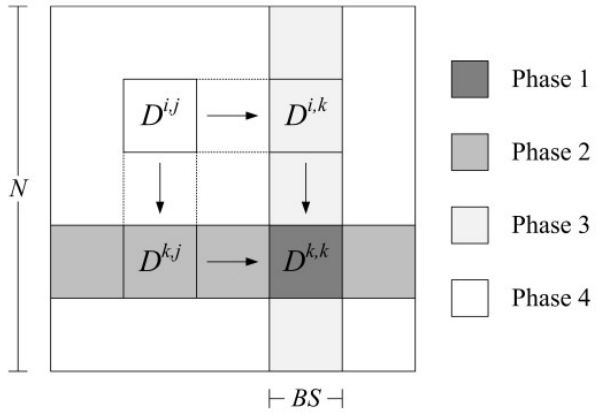

Fig 4.6: Etapas del algoritmo FW con blocking.

Fuente: Fig 2 [27, p. 4]

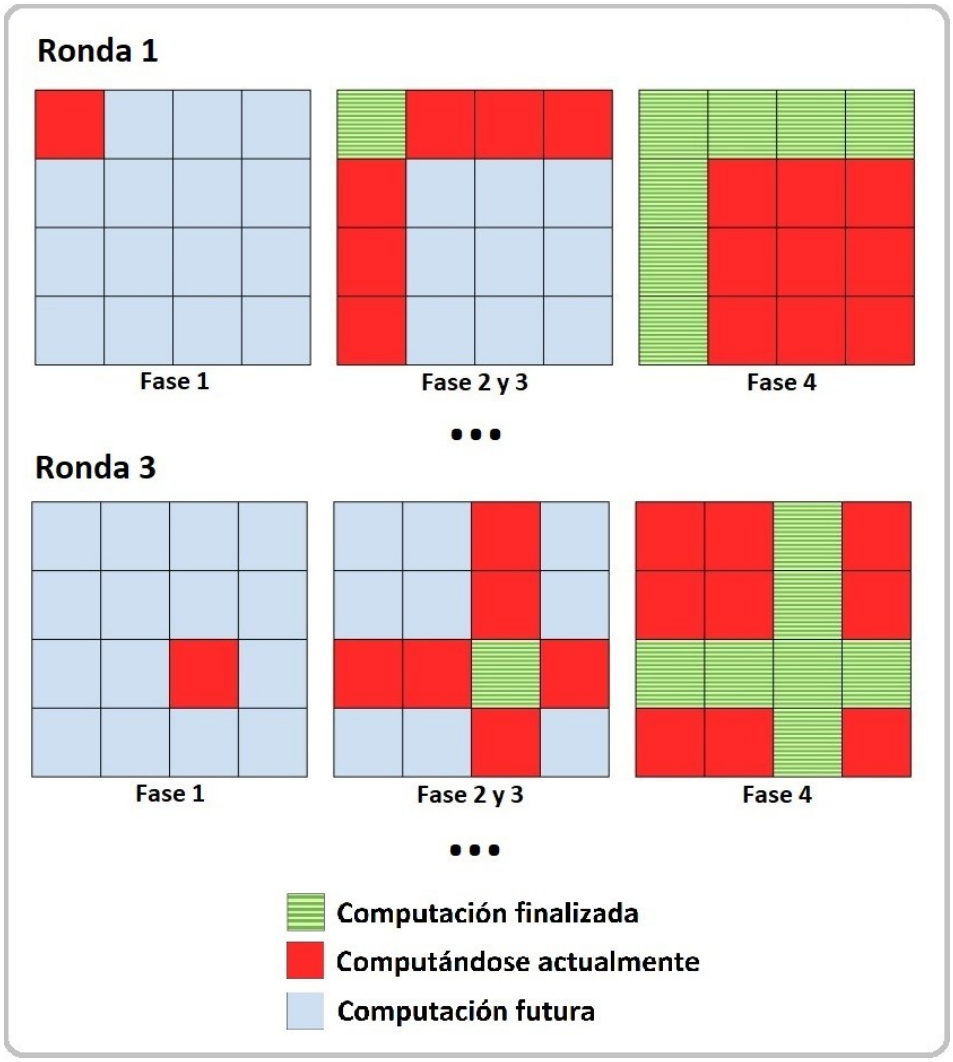

Fig 4.7: Etapas del algoritmo FW con blocking con fases 2 y 3 en paralelo.

Adaptación de Figura 1 [22]

Las fases 2 y 3 pueden ejecutarse en cualquier orden (como se grafica en la figura 4.7) sin corromper la correctitud del algoritmo; por lo que algunos autores la consideran en realidad una misma fase, no dos distintas. No obstante en este documento se las seguirá tratando como fases 2 y 3 por separado.

Pseudocódigo

```
1: function FW_BLOCK (D^1, D^2, D^3, P, base)
2:     for k = 0..BS-1 do
3:         for i = 0..BS-1 do
4:             for j = 0..BS-1 do
5:                 if D^2_{i,k} + D^3_{k,j} < D^1_{i,j} then
6:                     D^1_{i,j} = D^2_{i,k} + D^3_{k,j}
7:                     P_{i,j} = base+k
8:                 end if
9:             end for
10:        end for
11:    end for
12: end function

13: for k = 0..R-1 do
14:     b = k × BS
15:     # Phase 1
16:     FW_BLOCK(D^{k,k}, D^{k,k}, D^{k,k}, P^{k,k}, b)
17:     # Phase 2
18:     for j = 0..R-1, j ≠ k do
19:         FW_BLOCK(D^{k,j}, D^{k,k}, D^{k,j}, P^{k,j}, b)
20:     end for
21:     # Phase 3
22:     for i = 0..R-1, i ≠ k do
23:         FW_BLOCK(D^{i,k}, D^{i,k}, D^{k,k}, P^{i,k}, b)
24:     end for
25:     # Phase 4
26:     for i, j = 0..R-1, i,j ≠ k do
27:         FW_BLOCK(D^{i,j}, D^{i,k}, D^{k,j}, P^{i,j}, b)
28:     end for
29: end for
```

Fig 4.8: Pseudocódigo del algoritmo FW con blocking.

Fuente: Fig 3 [27, p. 5]

Descripción del pseudocódigo:

Función FW_BLOCK:

Como se puede observar en el pseudocódigo mostrado en la figura 4.8, lo que anteriormente era el algoritmo FW clásico con los tres *loops* anidados (ver figura 3.1), ahora se lo puede volver a encontrar en esta versión pero encapsulado en la función FW_BLOCK.

En la función FW_BLOCK, el *if* que antes efectuaba la comparación en base a tres valores de la matriz de distancia, en este caso dichos tres valores podrían pertenecen a tres bloques distintos. Por ese motivo, en esta función, en vez de recibir por parámetro una única matriz, ahora se reciben por parámetro tres matrices de distancias (D^1, D^2 y D^3), donde cada una representa un bloque, que podrían ser distintos entre sí.

Cabe destacar que los parámetros D^2 y D^3 son conmutativos, ya que internamente en la función solo se los suma.

Iteración entre rondas:

La función FW_BLOCK es llamada sucesivas veces durante las cuatro etapas de cada una de las R rondas. El primer *loop* (linea 13 de la figura 4.8) itera con el índice k sobre las R rondas*[1], y en cada una de estas rondas se ejecutan las 4 fases del algoritmo descriptas anteriormente.

Las variables i y j en este contexto representan el número de fila de bloques y el número de columna de bloques respectivamente.

Fases:

1. Simplemente ejecuta una única vez (por ronda) a la función FW_BLOCK, pasando el mismo bloque [k,k] a los parámetros D^1, D^2 y D^3, es decir, se procesa el bloque independiente de cada ronda.
2. Ejecuta r-1 veces la función FW_BLOCK procesando los bloques de la columna de bloques k, exceptuando el bloque [k,k] (ya procesado en la fase 1).
3. Ejecuta r-1 veces la función FW_BLOCK procesando los bloques de la fila de bloques k, exceptuando el bloque [k,k] (ya procesado en la fase 1).
4. Ejecuta $(r-1)^2$ veces la función FW_BLOCK procesando todos los bloques restantes; es decir, a todos los de la matriz, exceptuando a los de la columna de bloques k (ya procesada en la fase 2) y exceptuando la fila de bloques k (ya procesada en la fase 3).

Implementación en C

Al igual que en las anteriores implementaciones del algoritmo, nuevamente se guardan en variables los resultados parciales de los cálculos de los índices, eliminando así cómputo redundante.

Algo que cambia con respecto al pseudocódigo de la figura 4.8, es que no se reciben por parámetro las matrices (bloques de matrices) D^1, D^2 y D^3, sino que se reciben por parámetro los tres índices de la matriz D que apuntan al primer elemento de cada bloque D^1, D^2 y D^3. Se pueden ver las capturas de pantalla del código en las figuras 4.9 y 4.10

1 No confundir este k con el de la función FW_BLOCK, ya que en ese contexto representa al vértice intermedio de un camino, y aquí representa el número de ronda.

```
1 void floydWarshall(TYPE* D, int* P, int n){
2     INT64 i, j, k, r, b, kj, ik, kk, ij, row_of_blocks_disp, block_size, k_row_disp,
  k_col_disp, i_row_disp, j_col_disp;
3
4     r = n/BS;
5     row_of_blocks_disp = n*BS;
6     block_size = BS*BS;
7
8     for(k=0; k<r; k++){
9         b = k*BS;
10        k_row_disp = k*row_of_blocks_disp;
11        k_col_disp = k*block_size;
12
13        //Fase 1
14        kk = k_row_disp + k_col_disp;
15        FW_BLOCK(D, kk, kk, kk, P, b);
16
17        //Fase 2
18        for(j=0; j<r; j++){
19            if(j == k)
20                continue;
21            kj = k_row_disp + j*block_size;
22            FW_BLOCK(D, kj, kk, kj, P, b);
23        }
24
25        //Fase 3
26        for(i=0; i<r; i++){
27            if(i == k)
28                continue;
29            ik = i*row_of_blocks_disp + k_col_disp;
30            FW_BLOCK(D, ik, ik, kk, P, b);
31        }
32
33        //Fase 4
34        for(i=0; i<r; i++){
35            if(i == k)
36                continue;
37            i_row_disp = i*row_of_blocks_disp;
38            ik = i_row_disp + k_col_disp;
39            for(j=0; j<r; j++){
40                if(j == k)
41                    continue;
42                j_col_disp = j*block_size;
43                kj = k_row_disp + j_col_disp;
44                ij = i_row_disp + j_col_disp;
45                FW_BLOCK(D, ij, ik, kj, P, b);
46            }
47        }
48    }
49 }
```

Fig 4.9: Captura de pantalla del algoritmo FW con blocking (versión Block-Sec)

```
1  static inline void FW_BLOCK(TYPE* const graph, const INT64 d1, const INT64 d2, const INT64 d3,
   int* const path, const INT64 base){
2      INT64 i, j, k, i_disp, i_disp_d1, k_disp, k_disp_d3;
3      TYPE dij, dik, dkj, sum;
4
5      for(k=0; k<BS; k++){
6          k_disp = k*BS;
7          k_disp_d3 = k_disp + d3;
8          for(i=0; i<BS; i++){
9              i_disp = i*BS;
10             i_disp_d1 = i_disp + d1;
11             dik = graph[i_disp + d2 + k];
12             for(j=0; j<BS; j++){
13                 dij = graph[i_disp_d1 + j];
14                 dkj = graph[k_disp_d3 + j];
15                 sum = dik + dkj;
16                 if(sum < dij){
17                     graph[i_disp_d1 + j] = sum;
18                     path[i_disp_d1 + j] = base + k;
19                 }
20             }
21         }
22     }
23 }
```

Fig 4.10: Captura de pantalla de función FW_BLOCK (versión Block-Sec)

Organización de las matrices en memoria

Hasta ahora se detalló la implementación del FW con *blocking*, pero todavía no se ha dicho nada sobre cómo se guardan los datos (matrices) en los vectores. Como se verá a continuación, los vectores también sufrirán cambios para adaptarse al algoritmo con su nueva forma de recorrer las matrices, buscando así maximizar la localidad de datos en esta nueva solución.

A partir de la matriz representada anteriormente en la tabla 4.4, ahora con la técnica de *blocking* la matriz quedaría dividida en bloques como se representa en la tabla 4.6 con diferentes colores (cada color representa un bloque diferente).

[Fila, Columna]	**Columna 0**	**Columna 1**	**Columna 2**	**Columna 3**
Fila 0	[0,0]	[0,1]	[0,2]	[0,3]
Fila 1	[1,0]	[1,1]	[1,2]	[1,3]
Fila 2	[2,0]	[2,1]	[2,2]	[2,3]
Fila 3	[3,0]	[3,1]	[3,2]	[3,3]

Tabla 4.6: Matriz cuadrada de N=4 particionada en bloques de 2x2

Si se mantuviera la misma organización de la matriz en memoria a como estaba en el algoritmo FW “clásico”, es decir, ordenada por filas, el vector quedaría como se muestra en la tabla 4.7. Con esta organización, los bloques quedan evidentemente fragmentados y dispersos en el vector, comprometiendo así la localidad de datos, y por lo tanto, el rendimiento.

Hay que tener en cuenta que las filas internas de los bloques son leídas consecutivamente dentro del principal *hotspot* del programa (función FW_BLOCK), por lo que si se quiere evitar este cuello de botella en el rendimiento, se debe utilizar otra forma de ordenar la matiz en el vector.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
[0,0]	[0,1]	[0,2]	[0,3]	[1,0]	[1,1]	[1,2]	[1,3]	[2,0]	[2,1]	[2,2]	[2,3]	[3,0]	[3,1]	[3,2]	[3,3]

Tabla 4.7: Matriz cuadrada de N=4 particionada en bloques de 2x2 guardada en un vector siendo ordenada por filas.

Si bien la forma de ordenación de la matriz por filas en el vector era útil para el algoritmo FW "clásico", ya no resulta adecuada para esta versión con *blocking*, la cual cambia totalmente el patrón de accesos a memoria. Para solucionar este problema, se debe modificar el orden por uno basado en bloques, a modo de mejorar la secuencialidad en el acceso a los datos del vector.

Problema de la dispersión de filas de bloques en tamaño de problemas realistas:
A simple vista, mirando la tabla de ejemplo, puede parecer poco el *interleaving* de los fragmentos (filas) de bloques en memoria, pero esto cambia sustancialmente con tamaños de entrada realistas. Por ejemplo un bloque de 128x128, tendría cada fila separada en memoria en intervalos de 128 posiciones en el vector; y si este vector almacena por ejemplo números de 32 bits, entonces las filas de cada bloque quedarían dispersas en memoria en intervalos de 512 bytes, teniendo esto un impacto significativo en el rendimiento.

Si la matriz de la tabla 4.6 es guardada en el vector ordenada por filas de bloques, y a su vez los bloques son internamente ordenados por filas, entonces quedaría almacenada en memoria como se muestra en la tabla 4.8. De esta forma, se elimina el *interleaving* de las filas de bloques; es decir, todos los datos de un bloque quedan de forma contigua en memoria, facilitando así el acceso secuencial a la misma.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
[0,0]	[0,1]	[1,0]	[1,1]	[0,2]	[0,3]	[1,2]	[1,3]	[2,0]	[2,1]	[3,0]	[3,1]	[2,2]	[2,3]	[3,2]	[3,3]

Tabla 4.8: Matriz cuadrada de N=4 particionada en bloques de 2x2 guardada en un vector siendo ordenada por filas de bloques, y los bloques internamente siendo ordenados por filas.

Elección del tamaño de bloque óptimo

La elección del tamaño de bloque tiene un gran impacto en el rendimiento, ya que con este parámetro se manipula directamente la localidad espacial de datos. La idea es elegir un tamaño de bloque óptimo que aproveche de la mejor manera posible a la memoria caché de la CPU.

Si bien el análisis teórico no reemplaza al análisis empírico que se debe realizar para encontrar el valor de BS óptimo, por lo menos sí nos ayuda a establecer las cotas de un segmento de "probabilidad" de donde se encuentra el valor de BS óptimo real. Esto permite focalizar los valores de prueba de BS dentro de este segmento, evitando así efectuar demasiadas ejecuciones con valores de prueba muy lejanos. En pocas palabras, gracias al análisis teórico, sabremos que el BS óptimo real tiene muy poca probabilidad de encontrarse fuera de dicho segmento.

Cómo hallar el *working set*

Para poder calcular el BS óptimo teórico debemos hallar primero una fórmula aproximada del *working set* del programa. Ésta será una función dependiente del tamaño de bloque (parámetro BS).

Para hallar el *working set*, primero se debe calcular el espacio de memoria que ocupa individualmente un bloque de la matriz.

Fórmula para calcular el tamaño (en bytes) de un bloque de una matriz:

$$blockSize = \mathbf{BS^2} * \mathbf{size\_of\_data}$$

Siendo:

. ***size_of_data***: "Tamaño en bytes de un único elemento de la matriz"

. ***BS***: "Tamaño de bloque por lado" (es decir, cantidad de elementos por lado del bloque)

Por ejemplo, una matriz de floats (32 bits (4 bytes)) con ***BS***=64, tendrá bloques que ocupan individualmente en memoria $64^2 * 4$ bytes = 16384 bytes = 16KB.

Para calcular el *working set*, se tendrá en cuenta el tamaño en bytes de los bloques de las matrices y la cantidad máxima de bloques que pueden llegar a accederse en los *hotspots* del algoritmo .

El único hotspot en este caso es la función *FW_BLOCK()*. Ésta recibe por parámetro tres bloques que podrían ser distintos. Se puede observar que la función *floyd* "base" efectúa el llamado a *FW_BLOCK()* con dos bloques distintos en las fases 2 y 3, y con tres bloques distintos en la fase 4, por lo que queda confirmado que en el *working set* de este algoritmo hay tres bloques como peor caso. Vale aclarar que la fase 4 no solo es la que más datos demanda, sino también la que más cómputo consume.

Espacio de memoria extra a considerar dentro del *working set*:

. Espacio ocupado por variables y parámetros con tiempo de vida activo al momento de ejecutarse el hotspot.

. Variables y estructuras del sistema operativo que se encuentran en el contexto del proceso (se deberán tomar en cuenta en caso de llamados a *system calls* en pleno *hotspot*).

. Espacio ocupado por librerías utilizadas en el *hotspot*.

En el *hotspot* de FW no hay llamadas a funciones, ni a *system calls*, ni a librerías; por lo que podemos dejar al "espacio de memoria extra" como una constante, es decir, un valor independiente de N y de BS.

En base a lo especificado en los párrafos anteriores, podemos calcular el *working set* teórico aproximado del algoritmo con la siguiente fórmula:

$$workingSet = \mathbf{blockSize * 3 + C}$$

Siendo ***C*** la constante que representa el resto del espacio de memoria presente en el *working set*.

Si se reemplaza blockSize por sus componentes originales:

$$workingSet = \mathbf{(BS^2 * size\_of\_data) * 3 + C}$$

Como se trata de matrices de *floats* (32 bits (4 bytes)) la fórmula quedaría de la siguiente manera: $workingSet = \mathbf{BS^2 * 4 * 3 + C}$

Lo que se puede resumir en:

$$workingSet = \mathbf{BS^2 * 12 + C}$$

Claramente el *working set* es directamente proporcional al parámetro *BS*, *y* es por ese motivo que resulta tan importante elegir un buen *BS* para optimizar el uso de memoria caché.

Como se mencionó anteriormente, la palabra final a la hora de elegir el *BS* óptimo para ejecutar este algoritmo en una máquina y plataforma determinada se tendrá luego de efectuar un minucioso análisis empírico en dicha máquina. Es decir, se debe ejecutar el programa con distintos tamaños de *BS* y midiendo los tiempos resultantes. No obstante, primero hay que definir un conjunto de valores de BS de prueba, y precisamente es el análisis teórico el que nos ayuda a definirlo, acotándolo a un segmento de valores donde más probablemente se encuentre el BS óptimo real.

Comportamiento del algoritmo según el BS elegido

. Si se elije un *BS* chico: se reduce la localidad espacial. Si el *working set* es más chico que la memoria caché, se estaría desperdiciando capacidad de la misma, y por lo tanto, potencialmente desperdiciando rendimiento.

. Si se elije un *BS* grande: se incrementa la localidad espacial. Si el *working set* es más grande que la memoria caché, quedarán fuera de la misma datos que se acceden frecuentemente, teniendo ésto un impacto negativo en el rendimiento al aumentar significativamente los fallos de caché.

Cachés multinivel

En caso de cachés multinivel, ¿Cuál nivel se debe tener en cuenta a la hora de elegir los valores de *BS*?. La respuesta en simple, se deben tener en cuenta a todos los niveles de caché, pero principalmente se debe tener en cuenta el nivel más chico (L1D) y el nivel más grande (L2 en el caso del KNL), ya que serán los que delimitarán los extremos del segmento de valores de prueba de BS.

El siguiente paso para calcular el hipotético BS optimo ajustado a un nivel de caché específico es primero igualar el *working set* con la capacidad de dicho nivel de caché, lo que daría la siguiente expresión:

$$workingSet = \boldsymbol{L / H}$$

Siendo:
L: "Capacidad en bytes del nivel a tratar de la memoria caché"
H: "Cantidad de *threads* que comparten el mismo nivel de memoria caché"

> La división exacta de la capacidad de un nivel de caché entre los hilos que lo comparten en realidad no es una métrica precisa, ya que el *working set* de dos hilos diferentes podrían solaparse. No obstante, podemos utilizar este cálculo como una buena aproximación.

Si se reemplaza la variable *workingSet* por su fórmula:

$$\boldsymbol{BS^2 * 12 + C = L / H}$$

El siguiente paso es despejar BS:

$$BS^2 * 12 + C = \frac{L}{H} \Rightarrow BS^2 * 12 = \frac{L}{H} - C \Rightarrow BS^2 = \left(\frac{L}{H} - C\right) * \frac{1}{12} \Rightarrow BS^2 = \frac{L}{12H} - \frac{C}{12} \Rightarrow$$
$$\Rightarrow BS = \sqrt{\frac{L}{12H} - \frac{C}{12}}$$

Por lo que queda conformada la fórmula para calcular el BS óptimo teórico para un nivel determinado de memoria caché:

$$BS_{opt} = \sqrt{\frac{L}{12H} - \frac{C}{12}}$$

Dado que la constante C en esta implementación sería bastante chica, podríamos omitirla para simplificar los cálculos, por lo que la ecuación quedaría resumida de la siguiente manera:

$$\text{Si } C \approx 0 \implies BS_{opt} \approx \sqrt{\frac{L}{12\,H}}$$

Según qué valor se elija para BS, se podrá hacer uso óptimo de la caché de menor nivel (pero no óptimo de la caché de mayor nivel) si el valor de prueba está cerca del extremo izquierdo del segmento de valores de prueba; o bien podría hacer uso óptimo de la caché de mayor nivel (pero no óptimo de la caché de menor nivel) si está cerca del extremo derecho del segmento de valores de prueba. Si en su lugar se utiliza un valor intermedio del segmento, se utilizaría a ambas caches con un relativo buen grado de optimización, pero no tan bueno desde el punto de vista individual de cada una. Los valores de prueba que caen fuera del segmento serían los que teóricamente no hacen uso óptimo de ninguno de los niveles de caché, por lo que los dejamos como en una zona de "baja probabilidad" de que allí se encuentre el BS óptimo real.

4.4. Versión *Opt-0* y sus optimizaciones incrementales

4.4.1. Versión *Opt-0*: FW con *blocking* paralelizado

Para paralelizar el algoritmo FW con *blocking* visto en la sección anterior (4.3.3) se puede aplicar un enfoque de grano fino, de grano grueso, o bien uno mixto:

- Grano fino: se paralelizan los *loops* de la función *floyd* interna (***FW_BLOCK()***), por lo que se reparte el trabajo de cómputo de un bloque entre múltiples hilos (paralelización intra-bloque).
- Grano grueso: se paralelizan los *loops* que llevan a cabo cada una de las cuatro fases de cada ronda, por lo que se reparte el trabajo de cómputo de bloques enteros entre hilos (paralelización inter-bloque). Es decir, coloquialmente, "se reparten bloques" entre hilos.
- Mixto: se paraleliza el cómputo interno de los bloques (paralelización intra-bloque) y a su vez se distribuye el trabajo de bloques completos a entidades paralelas de procesamiento (paralelización inter-bloque).

NOTA: No es posible paralelizar el *loop* más externo, es decir, el que itera sobre las rondas (***r***), ya que ahí se mantienen las mismas dependencias entre iteraciones que se detallaron previamente.

Cabe destacar que en la fase 1 no se puede implementar una paralelización de grano grueso, ya que esta fase solo consta del procesamiento de un único bloque por ronda, es decir, no hay disponibles múltiples bloques para computar en paralelo. Si se implementara una solución de grano grueso "purista" implicaría ejecutar la fase 1 de forma secuencial por lo dicho anteriormente. Por este motivo resulta útil ejecutar al menos la fase 1 con paralelismo intra-bloque, independientemente de la estrategia utilizada en el resto de las fases.

Implementación

En la implementación de las pruebas de este trabajo se optó por emplear paralelismo de grano grueso, a excepción de la fase 1 que se ejecuta con paralelismo de grano fino. Para implementar toda paralelización a nivel de hilos se utilizó el estandar OpenMP.

```
1  void floydWarshall(TYPE* D, int* P, int n, int t){
2      INT64 r, row_of_blocks_disp, num_of_bock_elems;
3      r = n/BS;
4      row_of_blocks_disp = n*BS;
5      num_of_bock_elems = BS*BS;
6      #pragma omp parallel default(none)
   firstprivate(r,row_of_blocks_disp,num_of_bock_elems,D,P) num_threads(t)
7      {
8          INT64 i, j, k, b, kj, ik, kk, ij, k_row_disp, k_col_disp, i_row_disp, j_col_disp, w;
9
10         for(k=0; k<r; k++){
11             b = k*BS;
12             k_row_disp = k*row_of_blocks_disp;
13             k_col_disp = k*num_of_bock_elems;
14
15             //Fase 1
16             kk = k_row_disp + k_col_disp;
17             FW_BLOCK_PARALLEL(D, kk, kk, kk, P, b);
18
19             //Fase 2 y 3
20             #pragma omp for schedule(dynamic)
21             for(w=0; w<r*2; w++){
22                 if(w<r){ //Fase 2
23                     j = w;
24                     if(j == k)
25                         continue;
26                     kj = k_row_disp + j*num_of_bock_elems;
27                     FW_BLOCK(D, kj, kk, kj, P, b);
28                 } else { //Fase 3
29                     i = w - r;
30                     if(i == k)
31                         continue;
32                     ik = i*row_of_blocks_disp + k_col_disp;
33                     FW_BLOCK(D, ik, ik, kk, P, b);
34                 }
35             }
36
37             //Fase 4
38             #pragma omp for collapse(2) schedule(dynamic)
39             for(i=0; i<r; i++){
40                 for(j=0; j<r; j++){
41                     if( (j == k) || (i == k) )
42                         continue;
43                     i_row_disp = i*row_of_blocks_disp;
44                     ik = i_row_disp + k_col_disp;
45                     j_col_disp = j*num_of_bock_elems;
46                     kj = k_row_disp + j_col_disp;
47                     ij = i_row_disp + j_col_disp;
48                     FW_BLOCK(D, ij, ik, kj, P, b);
49                 }
50             }
51         }
52     }
53 }
```

Fig 4.11: Captura de pantalla de la función principal de FW paralelo con blocking (versión Opt-0)

En la captura mostrada en la figura 4.11, se puede observar la paralelización inter-bloque en las fases 2, 3 y 4, mediante el uso de las directivas *#pragma omp for*.

Las fases 2 y 3 fueron fusionadas en una sola, en un único *loop* paralelo. La principal finalidad de esta unión de fases, es evitar la merma de rendimiento que se produce con las combinaciones de parámetros que dejan a ***r*** (número de rondas) menor que ***T*** (cantidad de *threads*). Cuando esto ocurre, por ejemplo con ***r***=32 y ***T***=64, en la fase 2 solo 32 hilos tendrían trabajo; mientras, el resto de los hilos estarían ociosos; para luego ejecutar la fase 3 nuevamente con el mismo problema. Este comportamiento lleva a que la mitad de los núcleos (físicos) del procesador no tengan hilos activos que ejecutar.

> Recordar que el modelo 7230 de Xeon Phi utilizado para las pruebas tiene 64 núcleos físicos, por lo que resulta indispensable para el rendimiento que al menos 64 hilos estén activos en todo momento.

Ahora, con esta fusión de las fases 2 y 3 en un solo *loop*, hay ***r****2 iteraciones para distribuir entre los hilos, por lo que el ***r*** mínimo con el cual debajo de él se comienza a perder rendimiento en las fases 2 y 3 (por dejar hilos ociosos) pasa de 64 a 32.

También se puede observar en el código (figura 4.11) que los dos *for* anidados de la fase 4 fueron fusionados en un único *for* paralelo. Ésto, al igual que la fusión de las fases 2 y 3, también evita una perdida de rendimiento cuando ***r***<***T***. Por lo tanto, el ***r*** mínimo con el cual debajo de él se pierde rendimiento por dejar hilos ociosos en la fase 4, pasa de 64 a 8.

Funciones FW_BLOCK utilizadas en esta versión del algoritmo:

La función ***FW_BLOCK()*** es la misma de la versión secuencial del algoritmo con *blocking* (captura de pantalla en figura 4.10), esta procesa un bloque entero de forma secuencial. La función es utilizada en las fases que implementan paralelismo de grano grueso (fases 2, 3 y 4).

La función ***FW_BLOCK_PARALLEL()*** (captura de pantalla en figura 4.25) procesa un bloque de forma paralela (paralelismo intra-bloque). Es utilizada en la fase de paralelismo de grano fino (fase 1).

```
1 static inline void FW_BLOCK_PARALLEL(TYPE* const graph, const INT64 d1, const INT64 d2, const
  INT64 d3, int* const path, const INT64 base){
2     INT64 i, j, k, i_disp, i_disp_d1, k_disp, k_disp_d3;
3     TYPE dij, dik, dkj, sum;
4
5     for(k=0; k<BS; k++){
6         k_disp = k*BS;
7         k_disp_d3 = k_disp + d3;
8         #pragma omp for
9         for(i=0; i<BS; i++){
10            i_disp = i*BS;
11            i_disp_d1 = i_disp + d1;
12            dik = graph[i_disp + d2 + k];
13            for(j=0; j<BS; j++){
14                dij = graph[i_disp_d1 + j];
15                dkj = graph[k_disp_d3 + j];
16                sum = dik + dkj;
17                if(sum < dij){
18                    graph[i_disp_d1 + j] = sum;
19                    path[i_disp_d1 + j] = base + k;
20                }
21            }
22        }
23    }
24 }
```

Fig 4.12: Captura de pantalla de funcion FW_BLOCK_PARALLEL (versión Opt-0)

Elección del tamaño de bloque óptimo al utilizar SMT

Como se especificó en el capítulo 2 (ver sección 2.1.7), el procesador Xeon Phi KNL implementa la tecnología SMT, con la capacidad de ejecutar cuatro hilos por núcleo. Haciendo uso pleno de SMT en un KNL cada núcleo comparte la caché L1 entre 4 hilos, mientras que la L2 lo hace entre 8 hilos.

A la hora de calcular los límites del segmento de valores de BS de prueba, si se utiliza la tecnología SMT, hay que tener en cuenta la cantidad de *threads* que comparten cada nivel de memoria caché.

Repasando la fórmula:

$$BS_{opt} \approx \sqrt{\frac{L}{12\,H}}$$

Siendo:
L: “Capacidad en bytes del nivel a tratar de la memoria caché”
H: “Cantidad de *threads* que comparten el mismo nivel de memoria caché”

A partir de ahora, al agregarle la cantidad de *threads* a la ecuación, ya deja de haber un único segmento de valores de prueba en donde antes solo variaba BS. Ahora, se trataría más bien de una tabla de valores de prueba, como se muestra en la tabla 4.9. Es decir, ya no alcanza con buscar un único BS óptimo, ni de forma teórica ni empírica, ya que habrá uno por cada valor diferente de T. Lo que se debe encontrar ahora es la **combinación óptima de *BS* y *T***.

A continuación, en la tabla 4.9 se muestran las combinaciones de valores de BS y T a utilizar en las ejecuciones de las pruebas de este trabajo.

T	L1D		L2	
	Compartida e/ *threads*	Inicio del segmento (BS)	Compartida e/ *threads*	Fin del segmento (BS)
32*[1]	1	**52**	1	**296**
64	1	**52**	2	**209**
128	2	**37**	4	**148**
256	4	**26**	8	**105**

Tabla 4.9: Segmentos de valores de prueba de BS para los diferentes valores de T

Para los BS de prueba se utilizaran valores potencias de 2, de modo que las matrices (de tamaño también potencia de 2) queden perfectamente divididas en bloques iguales, evitando así cualquier perdida de rendimiento asociada a una asimetría en los tamaños de bloques.

En base a lo especificado en la tabla 4.9 y en los párrafos anteriores, los valores de BS a probar serán los siguientes: **32**, **64**, **128** y **256.**

Configuraciones a probar

BS: 32, 64, 128 y 256 para todo T.

T: 64, 128 y 256.

Configuración óptima

En la figura 4.13 se presentan los resultados de rendimiento obtenidos con los distintos valores BS para el T que resultó óptimo (256).

1 Ejecutar un programa en un Xeon Phi KNL con solo 32 hilos prácticamente nunca es la configuración ideal, ya que si bien en ese caso toda la memoria caché L2 de cada *tile* queda a plena disposición para el único hilo que albergado en cada *tile*, de esta forma se desperdicia la mitad de los núcleos disponibles, y por lo tanto, la mitad de los recursos de cómputo del procesador. Ésta es una configuración que prácticamente solo tiene sentido para programas que naturalmente no escalen al agregar más de 32 hilos, por limitación en su grado de paralelismo.

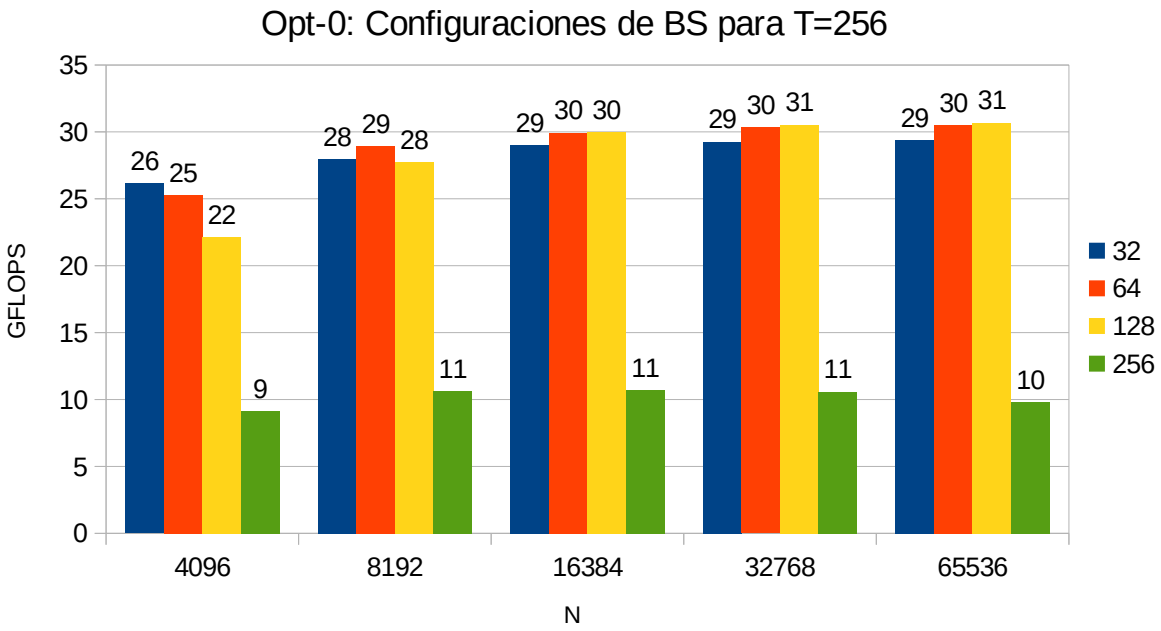

Fig 4.13: Versión Opt-0. Configuraciones de BS para T=256

En el gráfico se puede observar cómo el rendimiento cae con BS=256, al punto de llegar a un rendimiento similar al obtenido con la versión sin *blocking* del algoritmo (versión *Naive-Par* 4.3.2), rondando los 10 GFLOPS. Este acercamiento del rendimiento se debe a que a medida que el tamaño de bloque crece, la secuencia de pasos del algoritmo se termina pareciendo cada vez más a la versión sin *blocking*, por lo que resulta lógico que también se empiecen a parecer en términos de rendimiento.

En base a los resultados obtenidos, se deja asignada como configuración óptima de *BS* y *T* para esta versión del algoritmo a **BS=64** y **T=256**.

Resultados

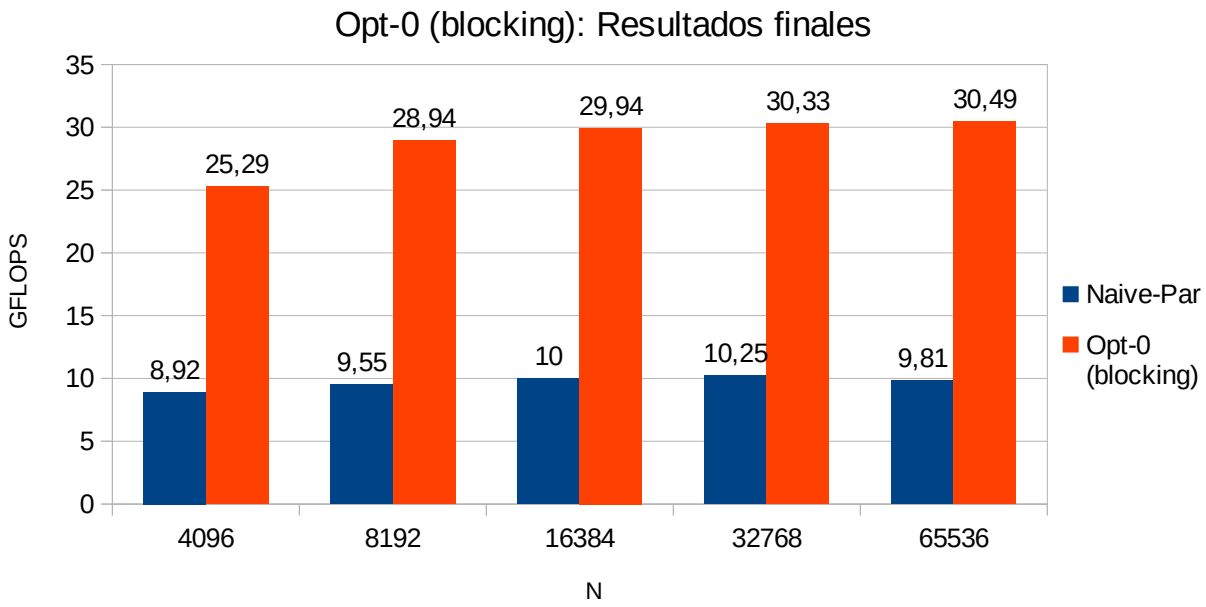

Fig 4.14: Resultados finales de la versión Opt-0

En esta versión efectivamente se solucionó el problema de la mala localidad de datos presente en la versión anterior (*Naive-Par*), por lo que ahora sí al aumentar la cantidad de hilos por núcleo se obtuvieron ganancias de rendimiento importantes. Esto se debe principalmente gracias a que los hilos ahora sí pueden trabajar de forma colaborativa con la caché L1D; por lo que al aumentar la cantidad de hilos, cada núcleo tiene más material de donde extraer instrucciones para llenar su *pipeline*; y de esta manera, se aprovechan mejor los recursos de cómputo.

La mejora promedio de esta versión fue de **2.99x**.

4.4.2. Versión *Opt-1*: Optimización utilizando MCDRAM

Implementación

Para utilizar la memoria MCDRAM se empleó el comando ***numactl*** (ya mencionado en la tabla 2.2) con el flag ***-p***, el cual hace que una vez completa la capacidad de la memoria MCDRAM, se empiece a asignar espacio en la memoria DDR; evitando así que se produzcan errores al tratar de utilizar más espacio del disponible en la MCDRAM (16GB). Esto es esencialmente útil para ejecutar este algoritmo FW con N=65536, con el cual el uso de memoria total supera los 32GB.

Configuraciones a probar

BS: 32, 64, 128 y 256 para todo T.

T: 64, 128 y 256.

Configuración óptima

Se mantuvo la misma configuración de BS y T óptima de la versión anterior; es decir, **BS=64** y **T=256**.

Resultados

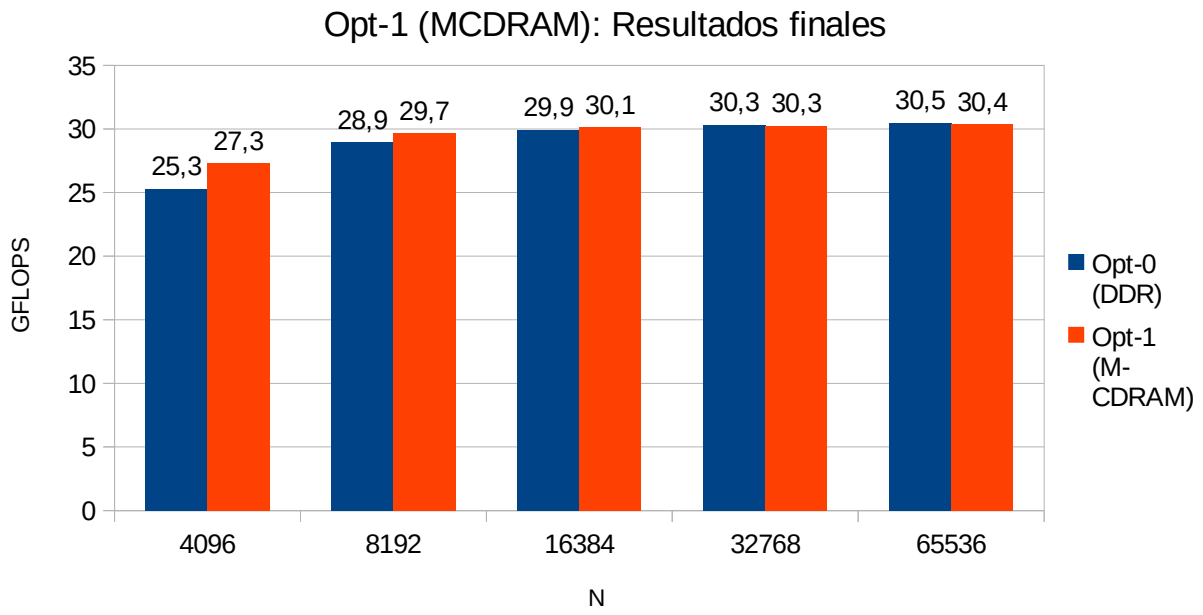

Fig 4.15: Resultados finales de la versión Opt-1

Como se puede observar en el gráfico comparativo de las versiones (figura 4.15), la ganancia de rendimiento al utilizar la memoria MCDRAM en este caso fue muy leve, de solo **1.02x**. No obstante, esto es esperable, ya que todavía en esta versión del programa no se tiene una gran demanda de ancho de banda de memoria, por lo que el brindado por la DDR resulta suficiente.

NOTA: Más adelante (sección 4.4.10) se hará una prueba ejecutando la versión *Opt-5* pero sin hacer uso la memoria MCDRAM. Allí, con un programa mucho más optimizado, se verá el potencial de la MCDRAM que aquí pasa desapercibido al no haber suficiente demanda de ancho de banda.

4.4.3. Versión *Opt-2*: Optimización utilizando vectorización guiada (SSE)

Implementación

Se implementa una vectorización guiada por directivas, haciendo uso de la directiva *#pragma omp simd* de OpenMP, tal como se puede apreciar en la capturas de pantalla del código fuente (figura 4.16). Como se vio previamente en la sección 2.3.6, con estas directivas se le indica al compilador que forzosamente vectorice un bucle, independientemente de lo que sus heurísticas le indiquen.

```
1  static inline void FW_BLOCK(TYPE* const graph, const INT64 d1, const INT64 d2, const INT64 d3,
   int* const path, const INT64 base){
2      INT64 i, j, k, i_disp, i_disp_d1, k_disp, k_disp_d3;
3      TYPE dij, dik, dkj, sum;
4
5      for(k=0; k<BS; k++){
6          k_disp = k*BS;
7          k_disp_d3 = k_disp + d3;
8          for(i=0; i<BS; i++){
9              i_disp = i*BS;
10             i_disp_d1 = i_disp + d1;
11             dik = graph[i_disp + d2 + k];
12             #pragma omp simd private(dij,dkj,sum)
13             for(j=0; j<BS; j++){
14                 dij = graph[i_disp_d1 + j];
15                 dkj = graph[k_disp_d3 + j];
16                 sum = dik + dkj;
17                 if(sum < dij){
18                     graph[i_disp_d1 + j] = sum;
19                     path[i_disp_d1 + j] = base + k;
20                 }
21             }
22         }
23     }
24 }
```

Fig 4.16: Captura de pantalla de la función FW_BLOCK (versión Opt-2)

Al igual que lo ocurrido en las versiones anteriores, la función FW_BLOCK_PARALLEL() queda similar a la función FW_BLOCK() (figura 4.16), pero con el agregado de la paralelización del bucle que itera sobre *i*. Por otro lado, la función *floydWarshall()* "principal" (la que llama a FW_BLOCK() y FW_BLOCK_PARALLEL()) queda igual a la versión anterior del algoritmo.

Configuraciones a probar

BS: 32, 64, 128 y 256 para todo T.

T: 64, 128 y 256.

Configuración óptima

En la figura 4.17 se presentan los resultados de rendimiento obtenidos con los distintos valores BS para el T que resultó óptimo (256).

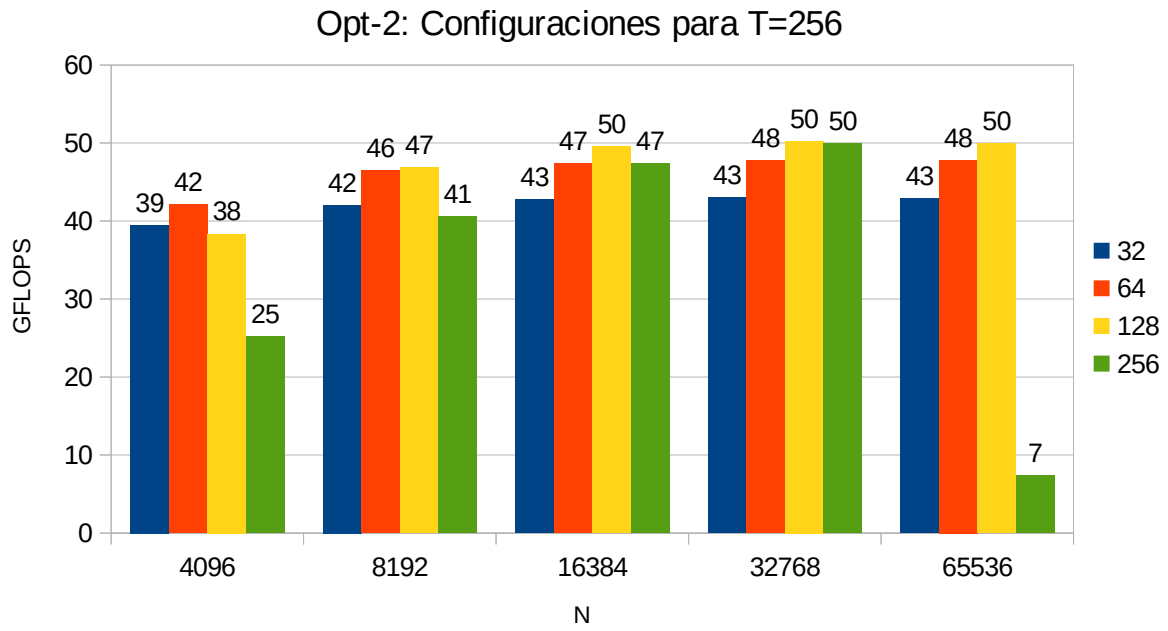

Fig 4.17: Versión Opt-2. Configuraciones de BS para T=256

La combinación óptima de BS y T en esta versión fue **BS=128** y **T=256**.

Resultados

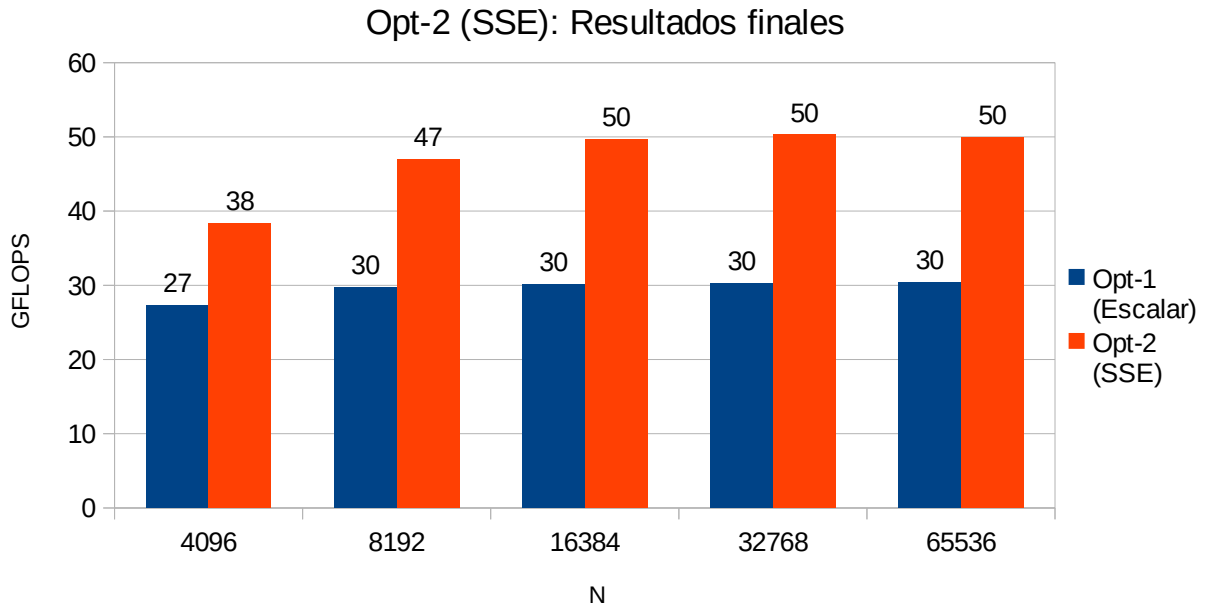

Fig 4.18: Resultados finales de la versión Opt-2

Esta versión vectorizada con SSE obtuvo una tasa de mejora promedio de **1.6x** con respecto a la versión anterior de cómputo escalar.

4.4.4. Versión *Opt-3*: Optimización utilizando vectorización guiada (AVX2)

Implementación

En este caso el código fuente se mantiene idéntico al de la versión anterior. Lo que cambia es que al compilador se le pasa por parámetro la opción ***-xAVX2***, para que en vez de generar código vectorizado con SSE, ahora lo haga con AVX2.

Configuraciones a probar

BS: 32, 64, 128 y 256 para todo T.

T: 64, 128 y 256.

Configuración óptima

En la figura 4.19 se presentan los resultados de rendimiento obtenidos con los distintos valores BS para el T que resultó óptimo (256).

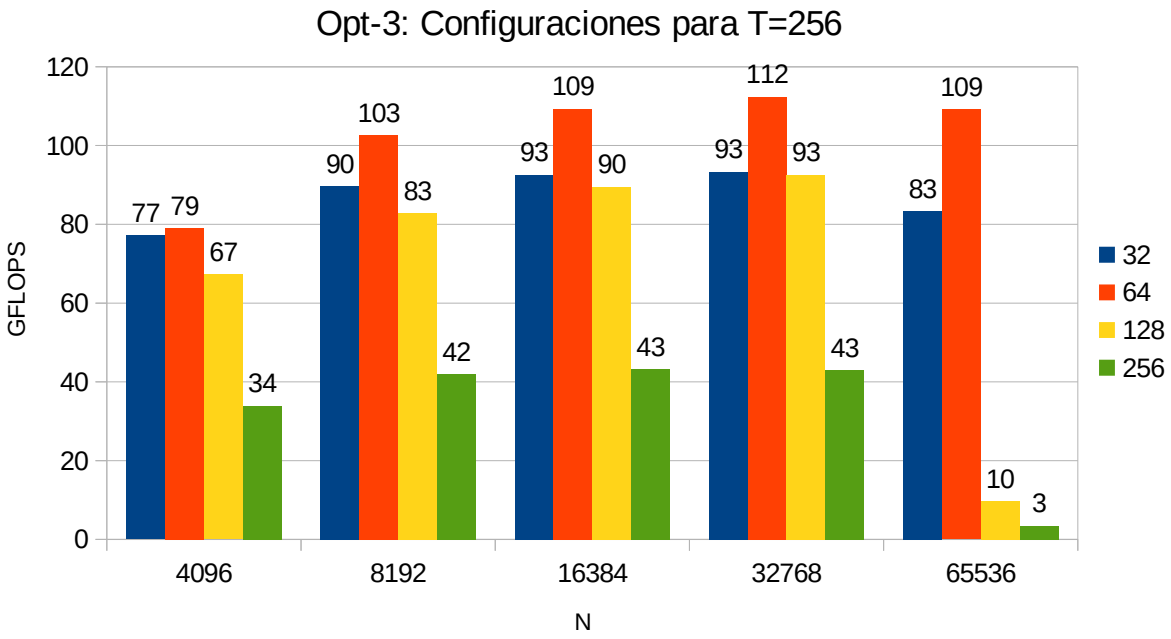

Fig 4.19: Versión Opt-3. Configuraciones de BS para T=256

La combinación óptima de BS y T en esta versión fue **BS=64** y **T=256**.

Resultados

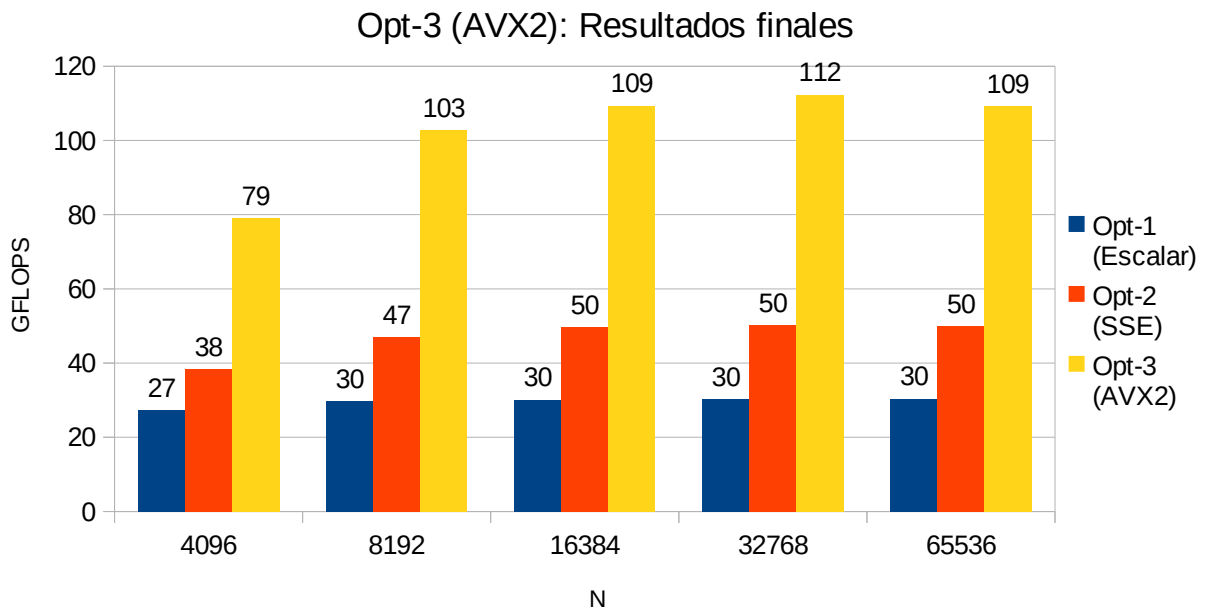

Fig 4.20: Resultados finales de la versión Opt-3

Esta versión vectorizada con AVX2 obtuvo una tasa de mejora promedio de **2.2x** con respecto a la versión anterior con SSE, y un acumulado de 3.5x con respecto a la versión *Opt-1* (sin vectorización).

4.4.5. Versión *Opt-4*: Optimización utilizando vectorización guiada (AVX512)

Implementación

Nuevamente, se mantiene el mismo código fuente de la versión anterior; pero esta vez pasando por parámetro al compilador la opción ***-xMIC-AVX512,*** de modo que en vez de generar código vectorizado con AVX2, ahora lo haga con AVX512.

Configuraciones a probar

BS: 32, 64, 128 y 256 para todo T.

T: 64, 128 y 256.

Configuración óptima

Con esta versión, el T óptimo pasó a ser 128, por lo que a continuación en la figura 4.21 se comparan los resultados obtenidos con los distintos BS para dicho T óptimo.

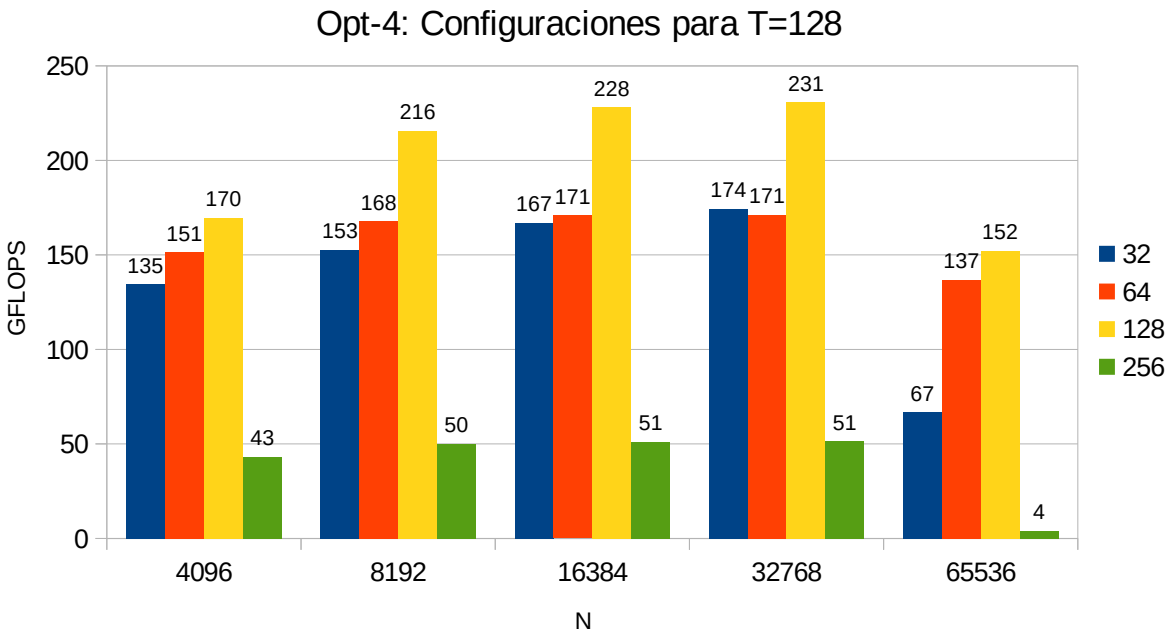

Fig 4.21: Versión Opt-4. Configuraciones de BS para T=128

La combinación óptima de BS y T en esta versión fue **BS=128** y **T=128**.

Resultados

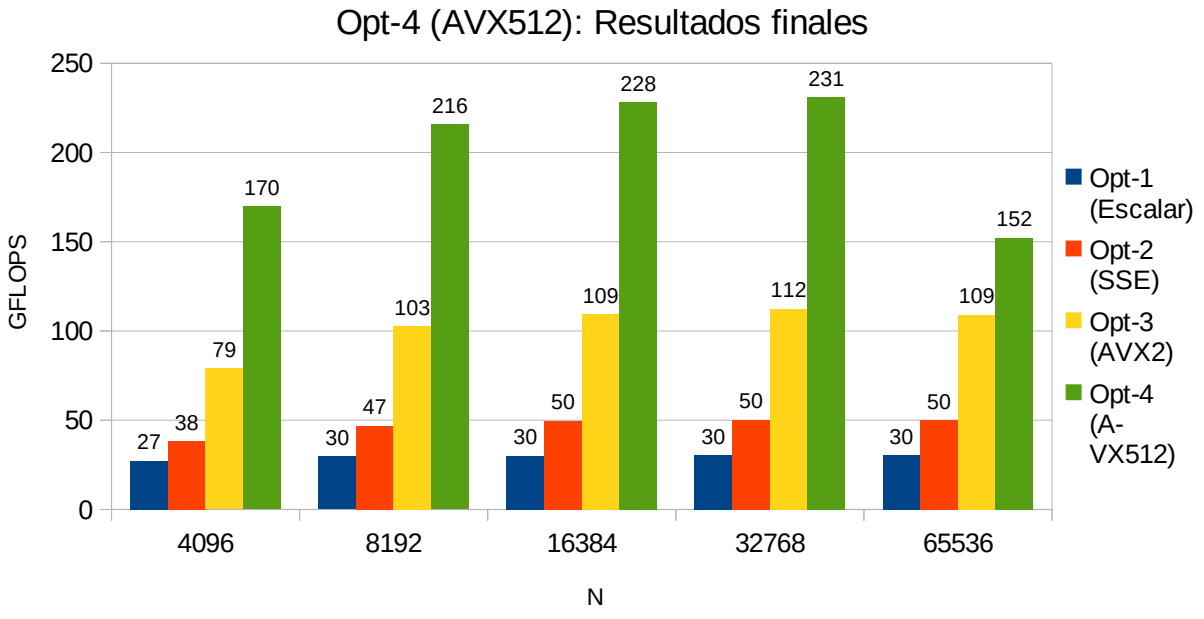

Fig 4.22: Resultados finales de la versión Opt-4

Esta versión vectorizada con AVX512 obtuvo una mejora promedio de **1.96x** con respecto a la

versión anterior que hacía uso de AVX2, y una tasa de mejora acumulada de 6.74x con respecto a la versión *Opt-1* (sin vectorización).

4.4.6. Versión *Opt-5*: Optimización utilizando alineación de datos en memoria

Implementación

Con respecto al código fuente de la anterior versión, en esta implementación solo se modifican las líneas donde se realiza la reserva de espacio en memoria dinámica. Más específicamente, se reemplaza cada llamado a la función *malloc()* por el llamado respectivo a la función *_mm_malloc()* provista por Intel, la cuál es un equivalente a la *malloc()* de la *glibc*, pero que implementa alineamiento de datos. Esta función requiere un parámetro extra, donde se le indica la cantidad de bytes con la que debe efectuar el alineamiento. Este número deberá coincidir con la cantidad de bytes que entran en una linea de caché de la L1D, que en el caso del KNL es de 64 bytes.

Acceso alineado a memoria

Con respecto al tipo de acceso a memoria (para mas detalle ver sección 2.3.7), se realizaron pruebas de rendimiento tanto asumiendo alineamiento como también mediante accesos genéricos a memoria. En una primera instancia, se probó utilizando el acceso a memoria genérico (sin modificaciones en el código ni directivas), y se tomaron los resultados de tiempo y GFLOPS para tener de referencia. Luego, se agregó la opción *aligned* a la directiva *simd* de *OpenMP*, la cual le indica al compilador que "puede confiar" en que los accesos a memoria que se realizarán en el bucle siguiente trabajarán siempre sobre datos perfectamente alineados. Como resultado, no solo se obtuvo menos rendimiento (del orden de -10% aproximadamente), sino que además, los reportes de vectorización del compilador informaron que no se implementaron accesos alineados a memoria en el bucle en cuestión. En última instancia, se probó reemplazando totalmente la directiva *simd* de OpenMP por la directiva de Intel *vector aligned*. Con esta directiva sí se logró que el compilador implemente accesos a memoria alineados, pero tampoco esto se tradujo en mejora alguna en el tiempo de ejecución; de hecho, se mantuvo la pérdida de rendimiento del orden del 10%. Estos escenarios probados con sus correspondientes resultados se resumen en la tabla 4.10.

Herramienta utilizada para implementar accesos alineados	**Código obtenido al compilar**	**Resultados de rendimiento**
Ninguna	Acceso genérico	El mejor
Opción "*aligned*" de la directiva "*simd*" de *OpenMP*	Acceso genérico	Pérdida del ~10%
Directiva "*vector aligned*" de Intel	Acceso alineado	Pérdida del ~10%

Tabla 4.10: Herramientas probadas para lograr accesos alineados a memoria

Luego de investigar más en profundidad, buscando el motivo del por qué los accesos alineados sobre datos alineados no mejoran el rendimiento, la respuesta yace en el diseño de la arquitectura del procesador. Intel a partir de su arquitectura Nehalem, no solo redujo fuertemente la perdida de rendimiento por acceder a datos no alineados en memoria, sino que además, directamente bajó a cero la penalización cuando se realizan accesos genéricos sobre datos correctamente alineados. De esta forma, con las últimas arquitecturas de Intel, deja de tener sentido la distinción entre accesos alineados y no alineados, debido a que ya no hay diferencia de rendimiento según "cómo se accede" a los datos; solo sigue siendo relevante para el rendimiento "cómo se guardan" estos en memoria (si de forma alineada o no alineada). Esto explica por qué con la opción *aligned* de *OpenMP* el compilador no implementó los accesos alineados, ya que este "es consciente" de que no hay ninguna necesidad de utilizarlos. La directiva *vector aligned* de Intel sí forzó al compilador a implementar dichos accesos alineados, gracias a una naturaleza más imperativa de la directiva. No obstante, en ninguno de los dos casos se obtuvo mejora alguna de rendimiento, ya que como se especificó anteriormente, la arquitectura tiene penalidad "cero" ante accesos genéricos sobre datos correctamente alineados.

Hasta ahora se explicó por qué el rendimiento no mejoró con los accesos alineados, solo queda por explicar el por qué de la pérdida del orden del 10% de rendimiento que estos conllevaron. Este comportamiento probablemente se deba a que estas directivas de alineamiento de datos sobre-escriben heurísticas del compilador, a menudo relacionadas con otras optimizaciones automáticas, no solo de alineamiento de datos. El problema con esto es que los parámetros elegidos automáticamente por el compilador suelen ser más eficaces para el rendimiento que los determinados por las directivas. Al final de cuentas, siempre la última palabra la tendrá el análisis empírico, ejecutando pruebas del programa haciendo uso de distintas directivas y parámetros hasta dar con la solución óptima, la cual en este caso, fue la de directamente no utilizar directivas de acceso alineado a memoria.

Configuraciones a probar

BS: 32, 64, 128 y 256 para todo T.

T: 64, 128 y 256.

Configuración óptima

Si bien en este caso hubieron picos de rendimiento distribuidos entre distintos valores de T, de todos modos a continuación se muestra solo el gráfico para T=128, ya que fue con éste donde se obtuvieron los picos de rendimiento para el tamaño de entrada más grande[1] (N=64K).

1 En HPC en general se prioriza la ganancia de rendimiento para los tamaños de entrada mas grandes.

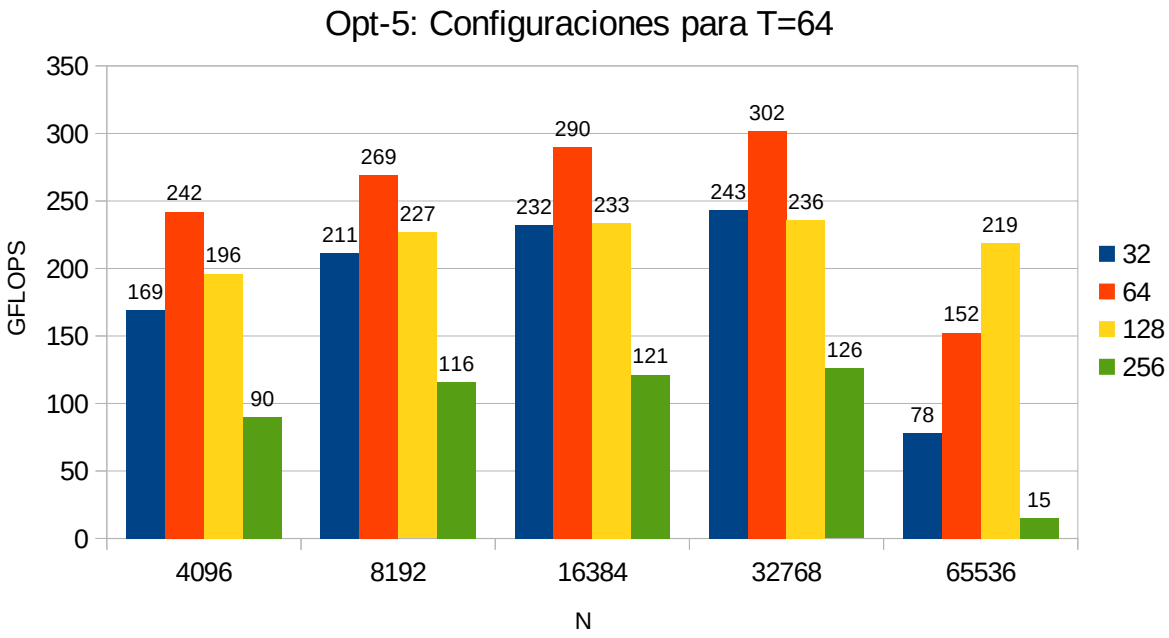

Fig 4.23: Versión Opt-5. Configuraciones de BS para T=64

La combinación óptima de BS y T en esta versión fue de **BS=128***[2] y **T=64**.

Resultados

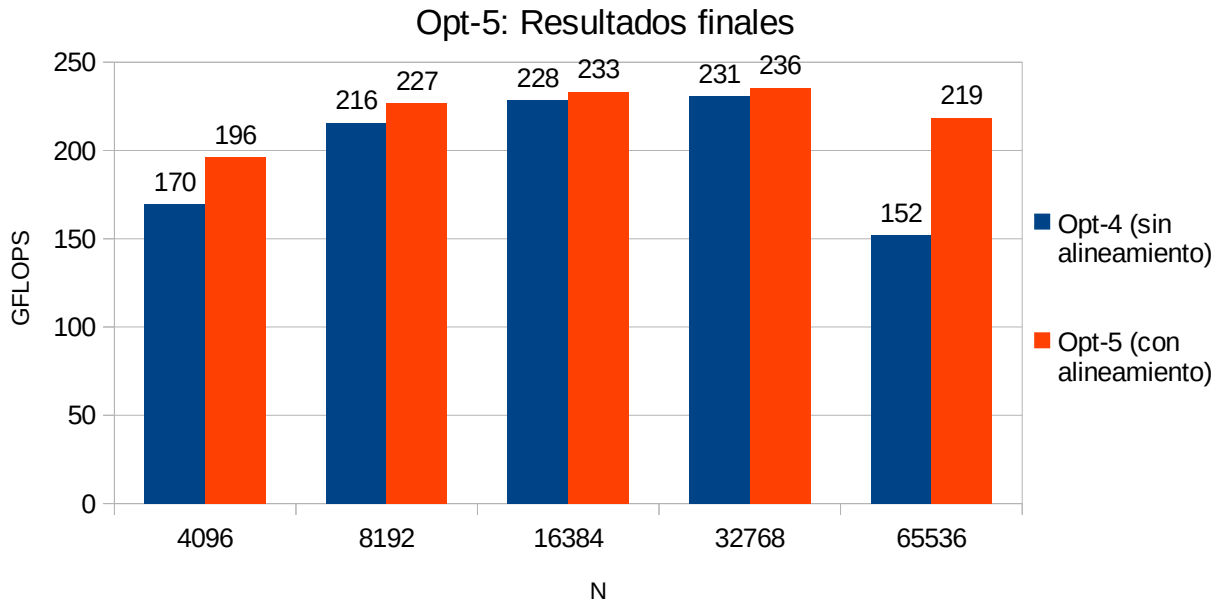

Fig 4.24: Resultados finales de la versión Opt-5

La tasa de mejora promedio de la versión fue de **1.14x**.

2 Se eligió BS=128 ya que fue con el que se obtuvo el mayor rendimiento para el tamaño de entrada más grande.

4.4.7. Versión *Opt-6*: Optimización utilizando predicción de saltos por software

Si bien en general la predicción de saltos automática de los procesadores funciona muy bien, hay situaciones en las que desde el punto de vista del software se puede saber de antemano cuál es la probabilidad de que un salto se efectúe. Si dicho salto condicional se encuentra en el *hotspot* de un programa, resulta importante ayudar al procesador a reducir al mínimo la cantidad de fallas en su predictor de saltos. Para lograr esto, los compiladores y lenguajes proveen diferentes herramientas con las cuales el programador puede dejar pistas en el código (pistas al compilador) sobre cuál es la probabilidad de que un salto se efectúe.

En el caso del algoritmo FW, independientemente de cualquier predicción de saltos y del grado de densidad del grafo, la condición del *IF* que tiene en su *hotspot*, en la mayoría de los casos da falso. Por este motivo, resulta importante asegurarse desde el software que el procesador no comience a ejecutar instrucciones del bloque del *IF* si todavía "no sabe" si la condición será verdadera o no.

Implementación

Se utiliza la función ***__builtin_expect(****long* ***exp,*** *long* ***c)****[1] soportada por GCC e ICC. Con ésta función se le indica al compilador cuál es el valor más probable (*c*) que se espera que arroje el resultado de la expresión (*exp*).

A partir de la función *__builtin_expect()* se pueden crear funciones más amigables, como las funciones *likely()* y *unlikely()* que se pueden observar en las líneas 1 y 2 del código mostrado en la figura 4.25. Con estas, la semántica se resume a que se espera *true* o *false* respectivamente, por lo que resultan ideales para utilizar en sentencias *IF*. En esta implementación, se probó tanto la función *likely()* como *unlikely()* en el *IF* más interno del algoritmo. No obstante, haciendo uso de *likely()* en dicho *IF*, el *assembler* generado fue idéntico al obtenido sin hacer uso de estas funciones (*Opt-5*), por lo que por defecto ya se estaba prediciendo la condición del IF como verdadera. Por este motivo se corrieron las pruebas de rendimiento solo haciendo uso de la predicción de salto negativa en el mencionado *IF* (utilizando la función *unlikely()*).

El código de las funciones *FW_BLOCK* de esta versión se puede observar en la figura 4.25. La función FW_BLOCK_PARALLEL es similar, solo que nuevamente tiene el agregado de la paralelización del bucle que itera sobre *i*. Por otro lado, la función FW base (la que llama a las funciones *FW_BLOCK*) se mantiene igual a la versión anterior.

1 Para más información sobre la función __builtin_expect: https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

```
#define likely(x)    __builtin_expect((x), 1)
#define unlikely(x) __builtin_expect((x), 0)

static inline void FW_BLOCK(TYPE* const graph, const INT64 d1, const INT64 d2,
const INT64 d3, int* const path, const INT64 base){
    INT64 i, j, k, i_disp, i_disp_d1, k_disp, k_disp_d3;
    TYPE dij, dik, dkj, sum;

    for(k=0; k<BS; k++){
        k_disp = k*BS;
        k_disp_d3 = k_disp + d3;
        for(i=0; i<BS; i++){
            i_disp = i*BS;
            i_disp_d1 = i_disp + d1;
            dik = graph[i_disp + d2 + k];
            #pragma omp simd private(dij,dkj,sum)
            for(j=0; j<BS; j++){
                dij = graph[i_disp_d1 + j];
                dkj = graph[k_disp_d3 + j];
                sum = dik + dkj;
                if(unlikely(sum < dij)){
                    graph[i_disp_d1 + j] = sum;
                    path[i_disp_d1 + j] = base + k;
                }
            }
        }
    }
}
```

Fig 4.25: Captura de pantalla de función FW_BLOCK (versión Opt-6)

Configuraciones a probar

BS: 32, 64, 128 y 256 para todo T.

T: 64, 128 y 256.

Configuración óptima

Al igual que en la versión anterior, no hubo un T óptimo universal para todo valor de BS. No obstante, a continuación en el figura 4.26 se muestran los resultados solo para T=128, ya que fue con éste donde se obtuvo más rendimiento para el tamaño de entrada más grande (N=64K).

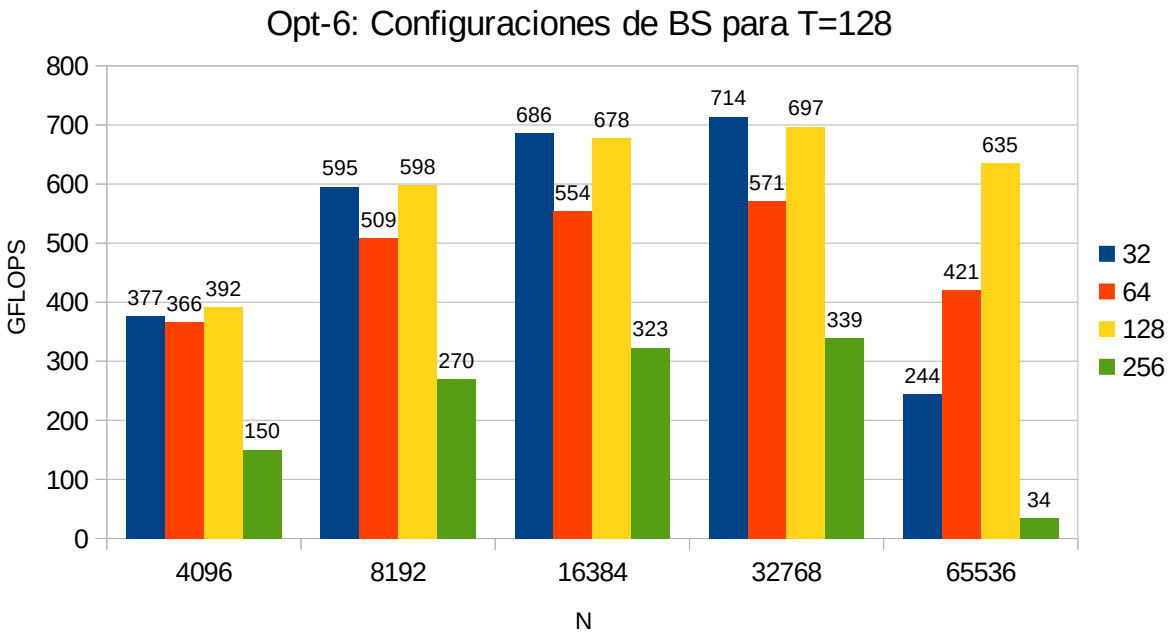

Fig 4.26: Versión Opt-6. Configuraciones de BS para T=128

Combinación óptima de BS y T en esta versión: **BS=128** y **T=128**.

Resultados

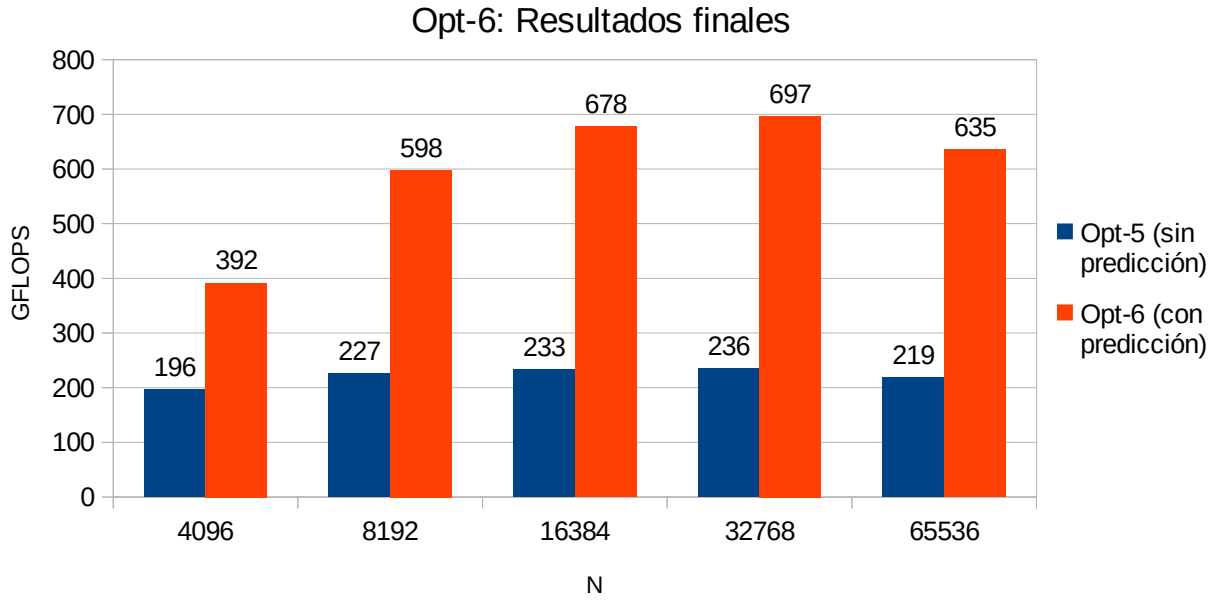

Fig 4.27: Resultados finales de la versión Opt-6

La mejora de rendimiento con esta optimización fue muy grande, y difícilmente se la pueda asociar únicamente a una mejor predicción de saltos por sí sola. Normalmente, una correcta predicción ayuda ahorrando solo algunos ciclos de reloj por cada instrucción de salto condicional, pero es improbable que por sí sola pueda duplicar el rendimiento de una aplicación (como es éste el caso para los valores de N más grandes). Aquí hay más detalles involucrados de fondo, los cuales permiten tal ganancia de rendimiento.

En este caso, una de las cuestiones indirectamente involucradas que complementan la predicción de saltos, es el *prefetching*. Normalmente, los procesadores modernos adelantan los accesos a memoria lo más que pueden (*prefetching*), de modo que cuando llega la instrucción que realmente accede a la memoria, ya el dato estaría leído, o al menos la línea de caché estaría en proceso de actualización, ocultando así parte del tiempo de respuesta en el acceso a memoria. El *prefetching* puede ser combinado agresivamente con predicción de saltos, permitiendo acceder a memoria de forma adelantada, aún cuando dichos accesos dependen de un salto condicional que todavía pudo no haberse resuelto. Esto presenta un arma de doble filo, ya que si la predicción de un salto falla, se habrán efectuado accesos a memoria totalmente inútiles. Dichos accesos "sin sentido" tienen un impacto negativo doble en el rendimiento; primero, aumentando la demanda de ancho de banda a memoria de manera ficticia (por lo que otros accesos a memoria, incluso de otros hilos, podrían verse perjudicados al competir por la memoria); y segundo, potencialmente trayendo líneas a caché que no serán utilizadas, dejando fuera a otras que probablemente sí sean parte del *woking set*.

Puntualmente en este algoritmo, lo que probablemente ocurría en la versión *Opt-5* es una mala predicción de saltos acompañada de un *prefetching* de escritura (llamado *prefetchW*). Este *prefeching* de escritura, si bien no escribe en memoria de forma adelantada (lo que rompería toda correctitud del algoritmo), sí carga en la caché las líneas correspondientes a las futuras escrituras en memoria, por más que estas luego solo se efectúen cuando la condición del IF sea verdadera (poco probable). En este caso, el problema más grande yace en la escritura en la matriz de reconstrucción de caminos mínimos (línea 23, figura 4.25), la cual es utilizada sólo dentro del *IF* (que generalmente es falso), por lo que se trata de una matriz que es accedida con una frecuencia muy baja (fuera del *working set*). En la versión *Opt-5*, este *prefetching* de escritura, en conjunto con la mala predicción de saltos, probablemente esté todo el tiempo trayendo a caché líneas correspondientes a la matriz de caminos, dejando fuera a líneas que albergan la matriz de distancias, la cual sí es accedida mucho más frecuentemente (parte del *working set*).

En síntesis, en esta versión, gracias a eliminar el anómalo *prefetching* de escritura (asociado a una mala predicción de saltos), se logró una mejora de rendimiento promedio de **2.68x** con respecto a la versión *Opt-5*.

4.4.8. Versión *Opt-7*: Optimización utilizando desenrollado de bucles

Implementación

En esta implementación se utilizó un desenrollado de bucles guiado por directivas*[1] para el bucle más interno (el que itera sobre j), y un desenrollado manual para el bucle intermedio (el que itera sobre i).

Desenrollado del bucle más interno

Se implementó un desenrollado de bucles guiado, haciendo uso de la directiva *unroll* soportada por el compilador de Intel. Esta directiva tiene como parámetro opcional el factor de desenrollado (FD), el cual en esta implementación es calculado exactamente para que el bucle quede completamente desenrollado, eliminando así todo salto condicional y toda instrucción de control asociada al mismo. Para lograr esto, se debe calcular (en tiempo de compilación) la cantidad de iteraciones del bucle en función del valor de BS, el ancho SIMD utilizado en la vectorización, y el tamaño del tipo de dato utilizado en el grafo.

Cómo obtener la cantidad de iteraciones del bucle:

cant = BS / (SIMD_WIDTH / TYPE_SIZE)

De base, en la versión *Opt-0*, el bucle en cuestión iteraba BS veces. Luego, con la vectorización implementada en las versiones *Opt-2*, *Opt-3* y *Opt-4*, la cantidad de iteraciones quedó reducida, en mayor o menor medida según el ancho de los registros SIMD. Particularmente en la versión *Opt-2*, haciendo uso de las instrucciones SSE que operan con registros de 128 bits, en cada iteración se procesan de a 4 números (128 bits / 32 bits (tamaño de *float*)), por lo que la cantidad de iteraciones queda reducida en BS/4. Luego, en la versión *Opt-3*, con el uso de instrucciones AVX2 de 256 bits, el bucle queda con BS/8 iteraciones, ya que en dicho caso se procesan de a 8 numeros (256 bits / 32 bits). Y por último, con AVX-512, luego del mismo cálculo, el bucle queda con BS/16 iteraciones.

Una vez obtenida la cantidad de iteraciones del bucle, ese mismo número será el FD a elegir para que dicho bucle quede totalmente desarmado. Con la macro definida en la linea 4 del código mostrado en la figura 4.28, se obtiene el FD de forma genérica para cualquier valor de BS, ancho SIMD y para cualquier tipo de dato.

Desenrollado del bucle intermedio

Adicionalmente, también se implementó un desenrollado manual del bucle intermedio (el que itera sobre i). En la captura del código en la figura 4.28 se puede apreciar como luego del desenrollado manual, el bloque del bucle fue repetido, mientras que sus iteraciones se redujeron a la mitad, sumándole 2 al índice luego de cada iteración.

1 Los distintos métodos de desenrollado de bucles fueron previamente explicado en la sección 2.3.8.

```
#define likely(x)    __builtin_expect((x), 1)
#define unlikely(x)  __builtin_expect((x), 0)

#define UNROLL_FACTOR BS/(SIMD_WIDTH/TYPE_SIZE)

static inline void FW_BLOCK(TYPE* const graph, const INT64 d1, const INT64 d2,
const INT64 d3, int* const path, const INT64 base){
    INT64 i, j, k, i_disp, i_disp_d1, k_disp, k_disp_d3;
    TYPE dij, dik, dkj, sum;

    for(k=0; k<BS; k++){
        k_disp = k*BS;
        k_disp_d3 = k_disp + d3;
        for(i=0; i<BS; i+=2){
            i_disp = i*BS;
            i_disp_d1 = i_disp + d1;
            dik = graph[i_disp + d2 + k];
            #ifdef INTEL_ARC
                #pragma unroll(UNROLL_FACTOR)
            #endif
            #pragma omp simd private(dij,dkj,sum)
            for(j=0; j<BS; j++){
                dij = graph[i_disp_d1 + j];
                dkj = graph[k_disp_d3 + j];
                sum = dik + dkj;
                if(unlikely(sum < dij)){
                    graph[i_disp_d1 + j] = sum;
                    path[i_disp_d1 + j] = base + k;
                }
            }
            i_disp = (i+1)*BS;
            i_disp_d1 = i_disp + d1;
            dik = graph[i_disp + d2 + k];
            #ifdef INTEL_ARC
                #pragma unroll(UNROLL_FACTOR)
            #endif
            #pragma omp simd private(dij,dkj,sum)
            for(j=0; j<BS; j++){
                dij = graph[i_disp_d1 + j];
                dkj = graph[k_disp_d3 + j];
                sum = dik + dkj;
                if(unlikely(sum < dij)){
                    graph[i_disp_d1 + j] = sum;
                    path[i_disp_d1 + j] = base + k;
                }
            }
        }
    }
}
```

Fig 4.28: Captura de pantalla de función FW_BLOCK (versión Opt-7)

Configuraciones a probar

BS: 32, 64, 128 y 256 para todo T.

T: 64, 128 y 256.

Configuración óptima

En la figura 4.29 se presentan los resultados de rendimiento obtenidos con los distintos valores BS para el T que resultó óptimo (128).

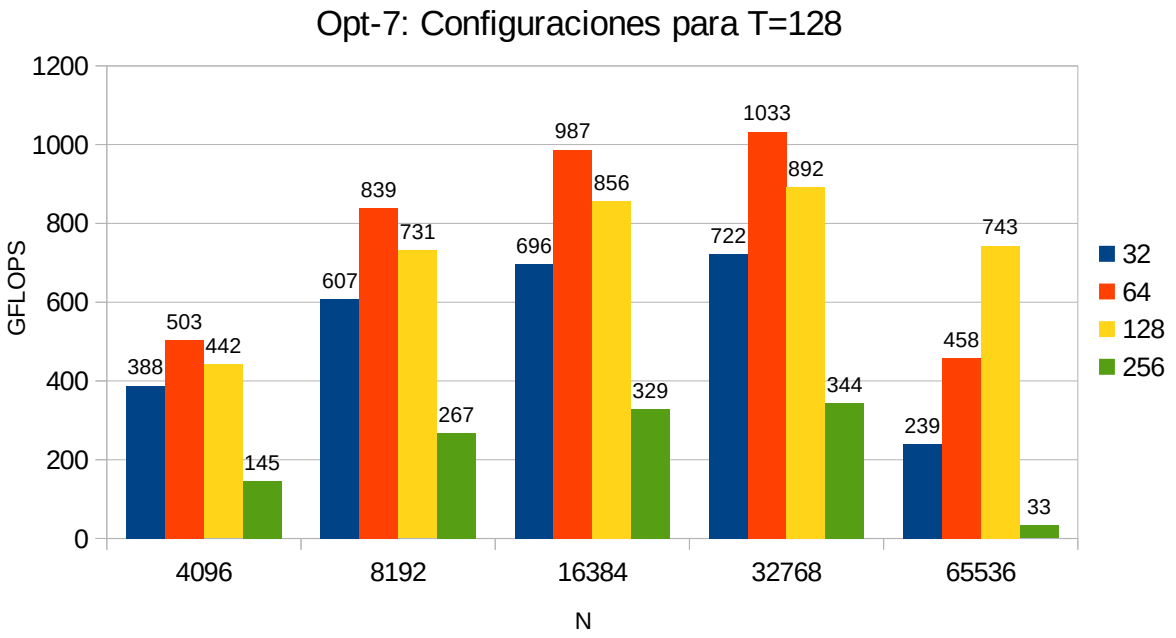

Fig 4.29: Versión Opt-7. Configuraciones de BS para T=128

Si bien el mejor BS para N=64K fue de 128, no obstante con éste se obtuvo bastante menos rendimiento para N<64K. Debido a esto, finalmente se deja al BS óptimo con dos valores distintos dependiendo de N: **BS=64** para N<64K, y **BS=128** para N=64K, ambos con **T=128**.

Resultados

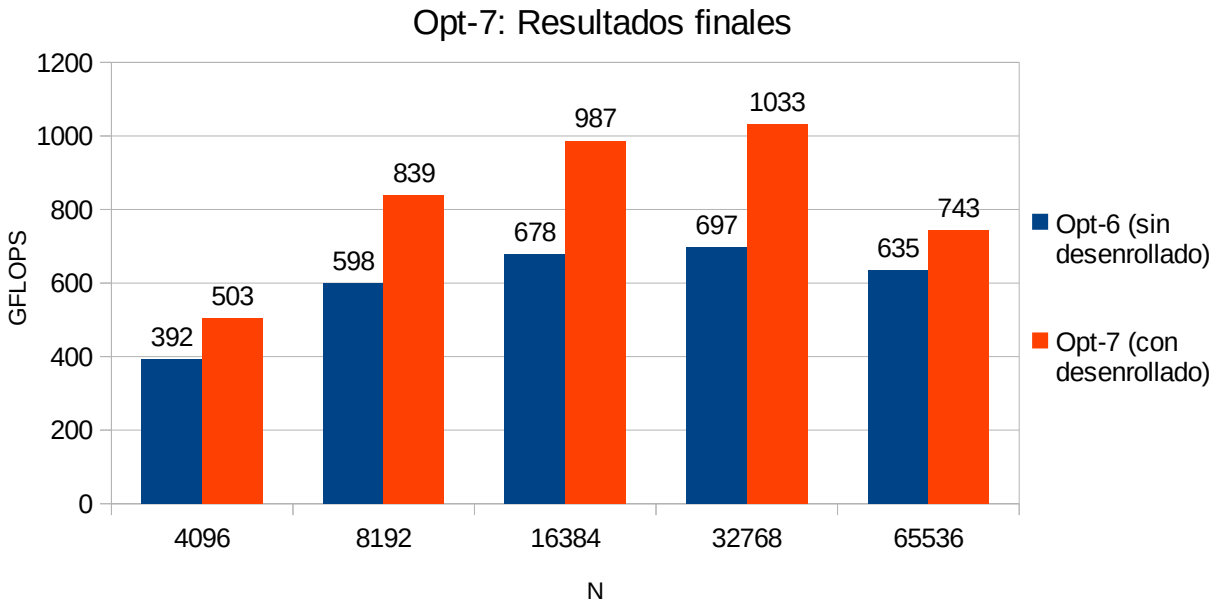

Fig 4.30: Resultados finales de la versión Opt-7

La tasa de mejora promedio de esta versión con respecto a la anterior fue de **1.36x**.

4.4.9. Versión *Opt-8*: Optimización utilizando afinidad de hilos con núcleos

Implementación

Para establecer una afinidad de hilos con núcleos se utilizó la variable de entorno KMP_AFFINITY, provista por la interfaz de afinidad de hilos de Intel. Las características básicas de la interfaz y su variable de entorno fueron previamente descritas en la sección 2.3.4.

Configuraciones a probar

Tipos de afinidad: *balanced* y *scatter**[1].
Granularidad: *fine* y *core*.

Configuración óptima

Al igual que en versión anterior, el T y BS óptimos se mantienen en T=128, con BS=64 para N<64K, y BS=128 para N=64K.

Las diferencias de rendimiento entre los distintos tipos de afinidad fue demasiado pequeña para ser distinguible en un gráfico de barras, por lo que los resultados son mostrados a continuación en una tabla (tabla 4.11).

Tabla 4.11: Versión Opt-8. Resultados de diferentes tipos de afinidad

N	Afinidad	Granularidad	
- - -	- - -	**fine**	**core**
4096	***sin afinidad**[2]*	502.82	
	balanced	506.18	475,96
	scatter	504.3	479.02
8192	***sin afinidad***	838.58	
	balanced	840	829,19
	scatter	836.49	830,98
16384	***sin afinidad***	986.87	
	balanced	987.67	982,49
	scatter	986.25	985.08
32768	***sin afinidad***	1032.65	
	balanced	1038.61	1032.29
	scatter	1036.32	1032.67
65536	***sin afinidad***	743.31	
	balanced	752.3	731.56
	scatter	753.02	743.32

1 La afinidad *compact* queda descartada por obtener sustancialmente peor rendimiento en todos los casos. Esto es debido a que este tipo de afinidad no es capaz de darle trabajo a todos los núcleos del procesador (como se describió previamente en la sección 2.3.4).

2 La afinidad ”*sin afinidad*” hace referencia al rendimiento obtenido con la anterior versión (*Opt-7*), la cual no establecía ninguna afinidad de hilos con núcleos.

La configuración óptima queda establecida en tipo de afinidad ***balanced*** con granularidad ***fine***.

Resultados

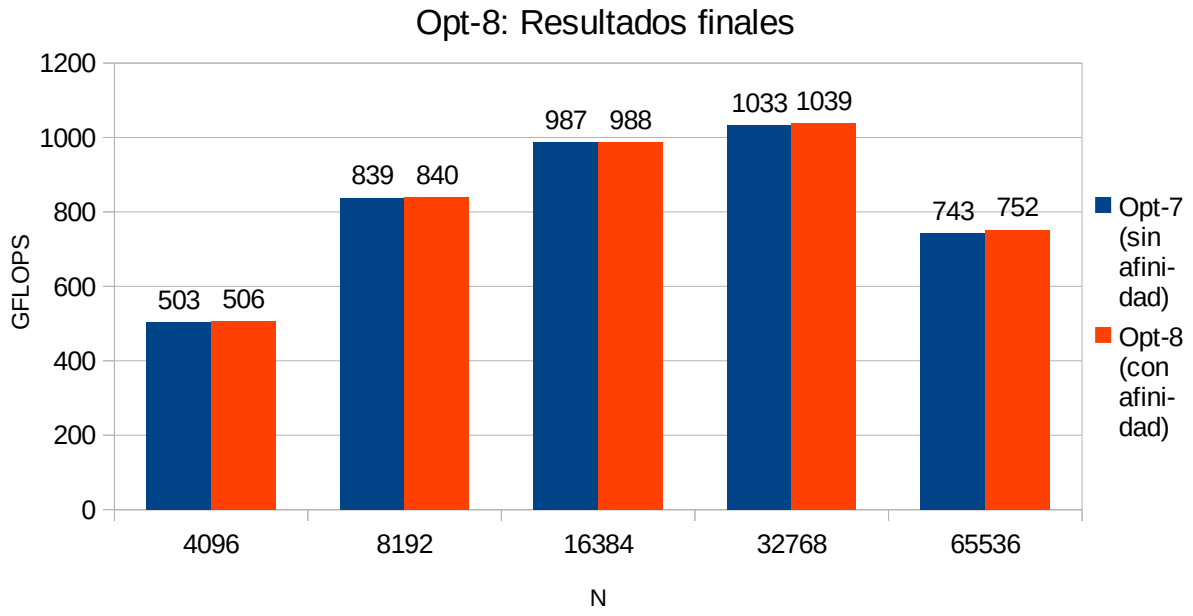

Fig 4.31: Resultados finales de la versión Opt-8

La tasa de mejora promedio de esta versión con respecto a la anterior fue de **1.005x**.

4.4.10. Probando eficacia de la MCDRAM sobre el nivel de optimización *Opt-5*

En este trabajo se utilizó por primera vez la memoria MCDRAM con la versión *Opt-1* (ver sección 4.4.2), donde se obtuvo una mejora de rendimiento de solo **1.02x**. Esta tasa de mejora fue tan escasa en ese entonces debido a la baja demanda de ancho de banda de memoria por parte del programa, el cual todavía tenía un muy bajo grado de madurez en sus optimizaciones. No obstante, luego las versiones *Opt-2*, *Opt-3* y *Opt-4* aumentaron fuertemente la demanda de ancho de banda al hacer uso intensivo de procesamiento SIMD, haciendo que cambien totalmente las características de la aplicación con respecto a su ejecución y uso de memoria. De esta forma, con el nivel de optimización *Opt-5*, si se deja de usar la memoria MCDRAM a cambio de utilizar puramente la DDR, es esperable una diferencia de rendimiento sustancial.

Para este experimento fue seleccionada la versión *Opt-5* de base y no la última (*Opt-8*) debido a que la *Opt-5* tiene mucha mayor demanda de ancho de banda, lo que permite exigir mucho más a las memorias, situación ideal para efectuar una correcta comparación entre ambas (MCDRAM versus DDR). La demanda de ancho de banda fue reducida a partir de la versión *Opt-6* con la predicción de saltos por software, la cual logró eliminar el problema que el *prefetching* de

escritura estaba causando en combinación con la mala predicción de saltos. De esta forma, se pudo eliminar la gran cantidad de operaciones de memoria no productivas que se estaban llevando a cabo.

Resultados

Para evaluar el rendimiento tanto con la MCDRAM como con la DDR se utilizó la configuración óptima de la versión base (*Opt-5*), es decir BS=128 y T=64.

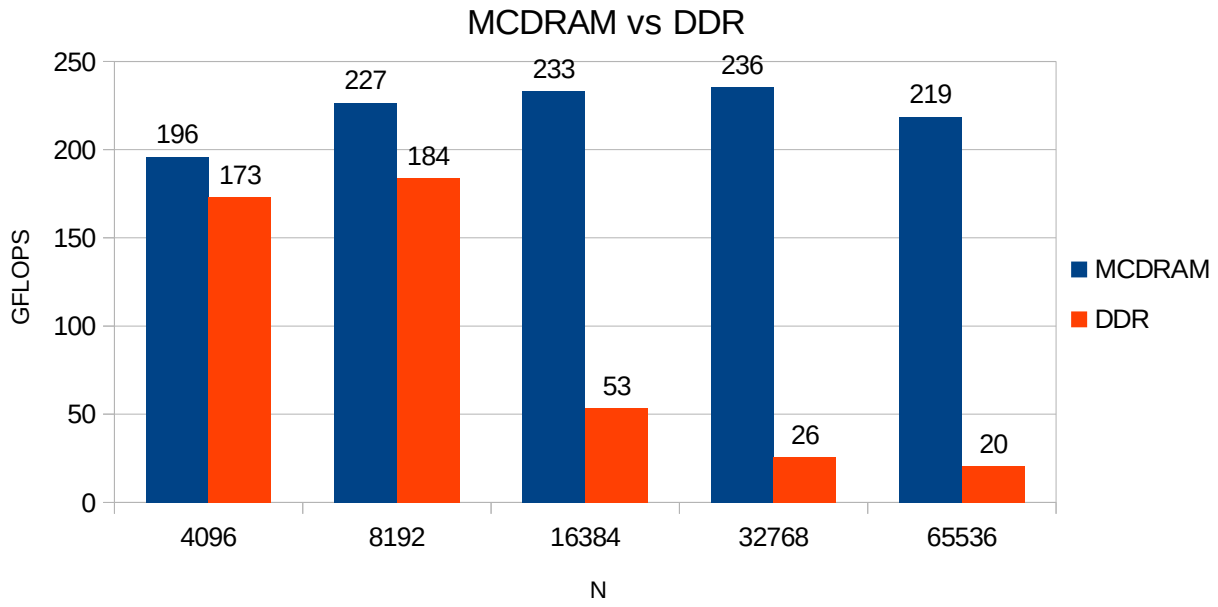

Fig 4.32: Comparativa de DDR vs MCDRAM utilizando de base la versión Opt-5

Con los resultados mostrados en la figura 4.32, se puede apreciar que efectivamente hubo una pérdida de rendimiento muy importante al dejar de utilizar la MCDRAM. Ahora sí queda en evidencia todo el potencial que la MCDRAM tiene para ofrecer con a su gran ancho de banda, el cuál no era "visible" todavía en la versión *Opt-1*, ya que allí éste no era aprovechado casi en absoluto.

Los resultados de GFLOPS fuertemente escalonados que se presentaron en este experimento se deben al uso de la memoria caché. Para grafos chicos (N=4K), el sistema de memoria pudo brindar un ancho de banda mayor que en el resto de los casos, ya que las matrices, en este caso al ser mucho más chicas, pudieron entrar en gran medida en las caches L2s combinadas. No obstante, a medida que dichas matrices crecen, cada vez una porción más chica de ellas entra en las caches, implicando una subida importante de la tasa de fallos de caché. Esto se ve agravado al no estar la MCDRAM satisfaciendo la gran demanda provocada por dichos fallos de caché, consolidándose de esta forma un intenso cuello de botella en el acceso a la memoria DDR.

Como lo indican los resultados de estos experimentos, la diferencia promedio de rendimiento que aportó la memoria MCDRAM con éste nivel de optimización (*Opt-5*) fue de **5.31x**.

4.5. Experimentos adicionales

4.5.1. Versión *Var-Nopath*: Variante que omite la matriz de reconstrucción de caminos mínimos

No siempre es necesario hallar la matriz de reconstrucción de caminos mínimos, en muchos casos, alcanza con calcular solamente la matriz de distancias mínimas. Si se omite el cómputo de la matriz de reconstrucción de caminos, se podría reducir sensiblemente el tiempo de ejecución total.

Implementación

La implementación de esta solución consta simplemente de comentar o eliminar la línea de código donde se actualiza la matriz de reconstrucción de caminos. Particularmente, las líneas 27 y 43 de la captura de pantalla de la figura 4.28.

Configuraciones a probar

BS: 32, 64, 128 y 256 para todo T.

T: 64, 128 y 256.

Configuración óptima

En la figura 4.33 se presentan los resultados de rendimiento obtenidos con los distintos valores BS para el T que resultó óptimo (128).

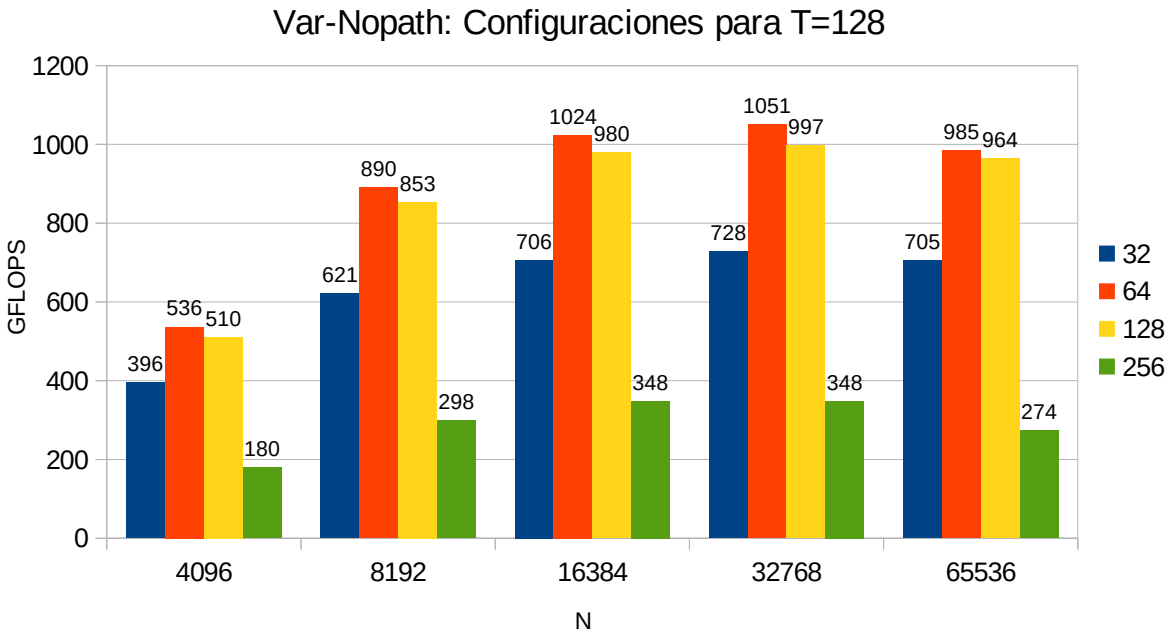

Fig 4.33: Versión Var-Nopath. Configuraciones de BS para T=128

Combinación óptima de BS y T en esta versión: **BS=64** y **T=128**.

Resultados

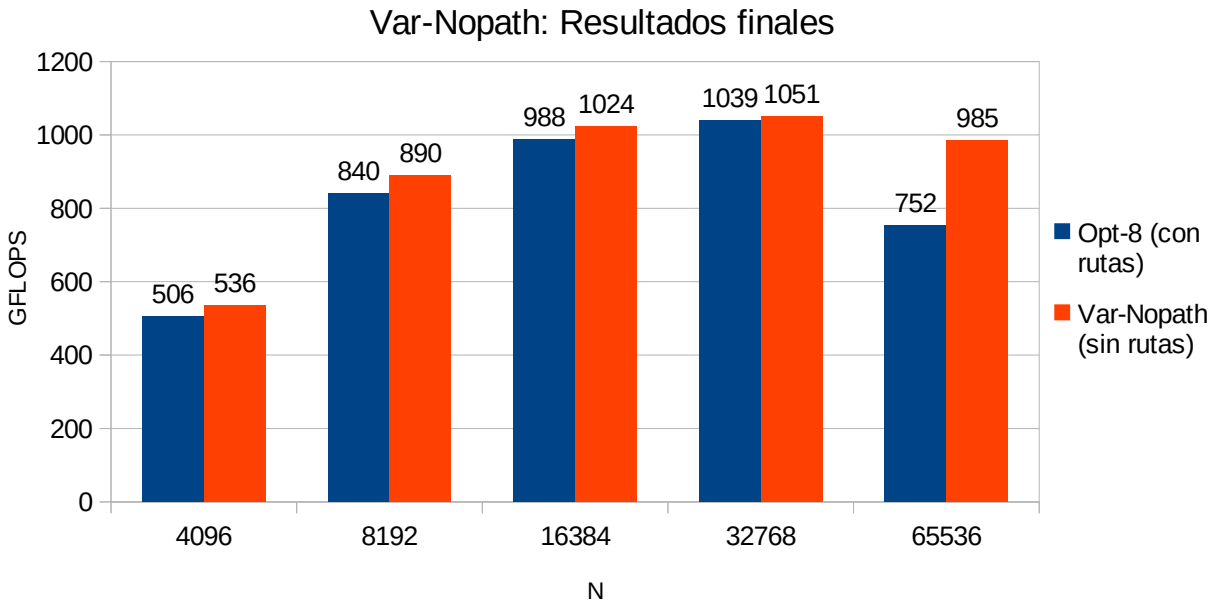

Fig 4.34: Resultados finales de la versión Var-Nopath

La mejora promedio de rendimiento de esta versión fue de **1.1x**. No obstante, vale la pena mencionar que puntualmente en el caso de N=64K, fue donde la reducción de los accesos a memoria de esta versión tuvieron su mayor impacto, logrando un mejora de 1.31x en dicho caso.

4.5.2. Versión *Var-Dens*: Variando el grado de densidad del grafo de entrada

Configuraciones a probar

Como es habitual en algoritmos de grafos, el tiempo de ejecución variará según el grado de densidad del grafo de entrada. Esto no es una excepción para el algoritmo FW, por lo que resulta interesante probar este programa con grafos de distintos grados de densidad.

GD (densidad de grafo): 0%, 2%, 5%, 10%, 20%, 40%, 80% y 100%.

Para simplificar el análisis, los valores de BS y T fueron fijados en BS=128 y T=128.

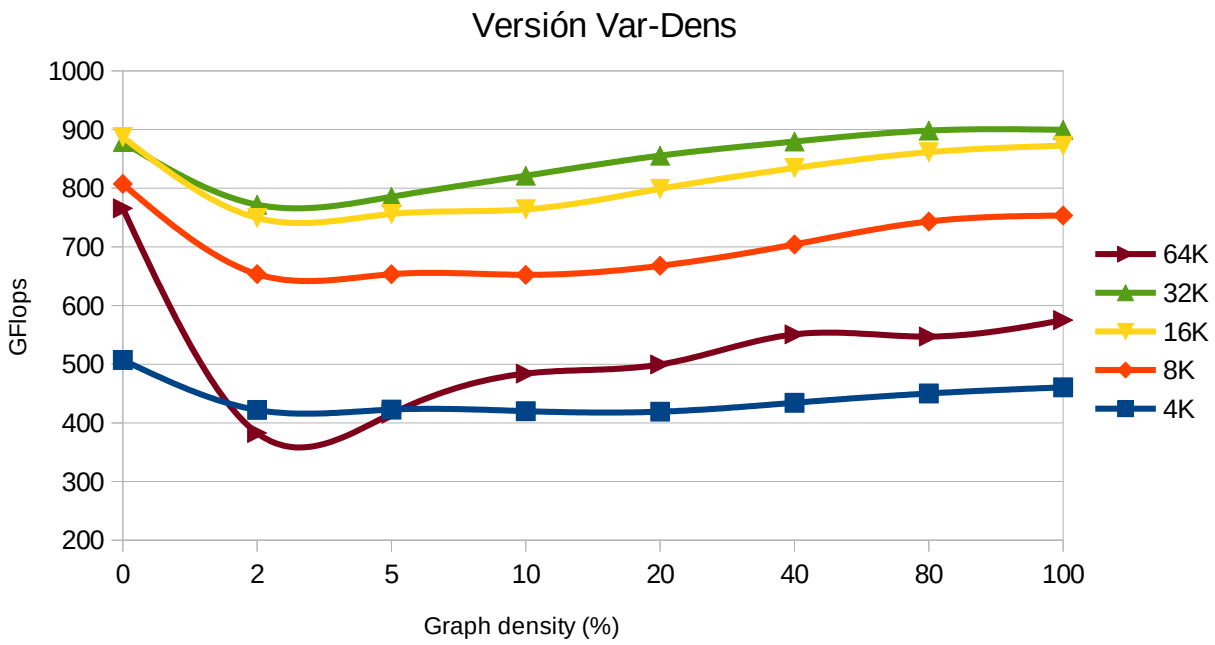

Fig 4.35: Versión Var-Dens: Rendimiento arrojado con distintos GD para cada valor de N

El mejor rendimiento, como era esperado, se obtuvo con la densidad en 0; es decir, con un grafo de entrada sin ninguna arista. La condición del *IF* siempre será falsa en este caso, por lo que la predicción de saltos tendrá una tasa de aciertos del 100%, situación óptima que conlleva al mejor rendimiento.

Cuando se agregan aristas al grafo, el rendimiento comienza a descender, ya que la tasa de aciertos de la predicción de saltos comienza a bajar. No obstante, a partir de una densidad del 2%, el rendimiento vuelve a subir a medida que dicha densidad también lo hace. Esto se debe probablemente a la naturaleza particular de los casos de prueba utilizados en este experimento, donde la longitud de los caminos del grafo tienden a ser mayores que el peso individual de cada arista. Es por esto que a partir de un punto, agregar más aristas solo hace que el peso de los caminos sea mayor, haciendo que sea poco probable que las aristas extra creen nuevos caminos mínimos (lo que haría verdadera a la condición del *IF*) más allá de los dos vértices que naturalmente conectan.

Si bien los GFLOPS tendieron a bajar con GD=2%, de todos modos las curvas para los distintos N presentaron pocas variaciones, manteniendo un rendimiento parejo. Esto se debe a la naturaleza del algoritmo FW, en la cual el orden del tiempo de ejecución es $O(|V|^3)$, es decir, es independientemente de la cantidad de aristas, y por lo tanto, independiente del grado de densidad del grafo.

4.5.3. Versión *Var-Prec*: Variante con relajación de precisión

Si en el contexto en que el programa va a ser utilizado no se requiere precisión fina en los decimales del peso de las aristas, entonces se le puede indicar al compilador que realice optimizaciones adicionales a las operaciones de punto flotante, a costa de posiblemente perder precisión en el resultado.

Implementación

A la hora de compilar el código se pasa al compilador (ICC) el siguiente parámetro: "*–fp-model fast=2*".

Resultados

El *assembler* resultante al compilar esta versión fue idéntico al de la versión base (*Opt-8*), por lo que el parámetro pasado al compilador no tuvo efecto alguno en este caso. Como en el algoritmo FW solo se realizan sumas y comparaciones de *floats*, el compilador no implementa ninguna optimización extra a estas operaciones. Esto se debe a que la suma aritmética entre *floats* ya de por sí es una operación rápida, por lo que no hay muchas más optimizaciones que el compilador pueda hacerles. Aparentemente el nivel de optimización 2 para operaciones de punto flotante se centra más bien en optimizar las operaciones que involucren multiplicaciones y divisiones, las cuales están totalmente ausentes en este caso.

No hubo ninguna mejora de rendimiento apreciable, por lo que esta optimización queda descartada.

4.5.4. Versión *Var-Double*: Variante con tipo de dato *double*

Introducción

Los distintos tipos de dato implican diferentes cargas de trabajo a los procesadores, por lo que de ser posible, siempre se deben evitar los tipos de datos que demandan más cómputo si éstos no son realmente necesarios. No obstante, hay dominios donde dichos tipos de datos "más pesados" son requeridos, por lo que resulta interesante comparar el rendimiento obtenido entre el tipo de dato *float* y *double*, su homólogo de 64 bits.

Configuración óptima

La combinación óptima de BS y T utilizando el tipo de dato *double* fue de T=64, con BS=64 para N<64K, y BS=128 para N=64K.

Resultados

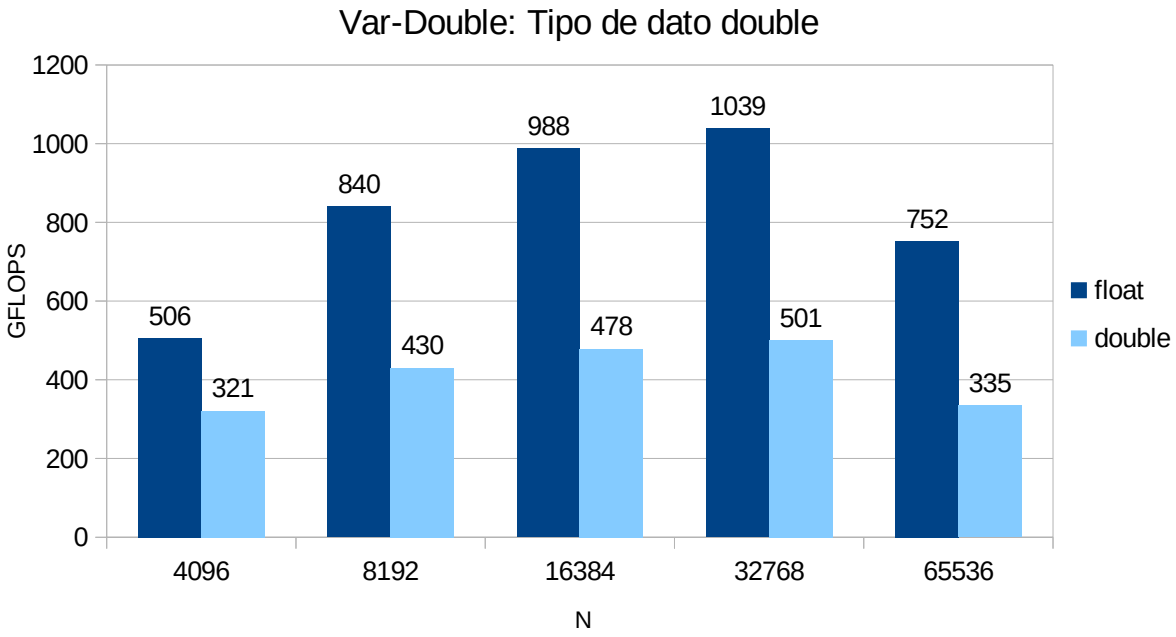

Fig 4.36: Versión Var-Double. Gráfico de comparación de rendimiento float vs double

Como se puede apreciar en los resultados (figura 4.36), con el tipo de dato de 32 bits (*float*) se obtiene aproximadamente el doble de rendimiento que con su correspondiente versión de 64 bits (*double*). Este comportamiento es el esperado, ya que en los registros AVX-512, donde entran sólo 8 datos de 64 bits, en su lugar entran 16 de 32 bits, permitiendo así que cada núcleo del procesador pueda computar el doble de números en simultáneo. Además, también en caché y en memoria MCDRAM se pueden alojar el doble de números que con 64 bits, siendo estos factores clave que permiten doblar el rendimiento con el tipo de dato de 32 bits.

4.6. Comparación con trabajos similares

Como se mencionó en la sección 3.3, existen numerosas propuestas de soluciones paralelas para FW. Sin embargo, en esta sección, sólo se considera la de Rucci et al. [27], ya que es la única que emplea aceleradores KNL como arquitectura de soporte. Al igual que en esta investigación, los autores estudiaron las diferencias de rendimiento al utilizar los distintos conjuntos de instrucciones SIMD, el uso de alineación de datos en memoria, desenrollado de bucles, y por último, los distintos tipos de afinidad de hilos con núcleos. A diferencia de este trabajo, en esta tesina se implementaron también otras optimizaciones, como la paralelización intra-bloque de la fase 1 de FW (con *blocking*), y la predicción de saltos por software. Resulta importante mencionar que, si bien el modelo de Xeon Phi KNL y el compilador usados son diferentes, en esta tesina se logró superar ampliamente el pico de rendimiento conseguido por esta propuesta previa, con diferencias de hasta 3x. En forma complementaria, también se agregaron otros experimentos adicionales, probando el rendimiento para distintos grados de densidad del grafo

de entrada; luego comparando el rendimiento entregado entre el tipo de dato *double y float*; y por último, evaluando el rendimiento de la variante de FW que omite el cálculo de la matriz de reconstrucción de caminos mínimos.

4.7. Resumen

El algoritmo FW "clásico", con su paralelización simple, apenas alcanzó los 10 GFLOPS al ser corrido en el KNL, particularmente tardando 16 horas en ejecutarse para N=64K. Si bien este rendimiento fue muy bajo, no obstante sirvió como referencia y punto de partida, además de llevar la mera distinción de ser la versión más sencilla de implementar en general.

El programa sufrió una fuerte reimplementación con la versión *Opt-0*, trayendo con ésta las técnicas de *blocking* al algoritmo FW con el objetivo de hacer un uso más eficiente de la memoria caché. Con el FW "clásico" se tiene el problema de una muy mala localidad de datos en memoria, con bucles recorriendo amplios espacios de la memoria. Es justamente esto lo que soluciona la versión *Opt-0*, la cual acota dichos bucles según un tamaño de bloque determinado (BS). A partir de la versión *Opt-0*, variando el BS efectivamente se puede ajustar el *working set* del programa hasta lograr un uso óptimo de la memoria caché. Con esta reimplementación del algoritmo (versión *Opt-0*), se logró triplicar el rendimiento con respecto a la versión sin *blocking*, logrando así unos 30 GFLOPS.

En la versión *Opt-1*, haciendo uso de la memoria MCDRAM, prácticamente no se obtuvo mejora alguna en el rendimiento. Esto se debe a que todavía en dicho nivel de optimización la poca demanda de ancho de banda es satisfecha por completo por la memoria DDR en solitario. No obstante, más adelante, se experimentó deshabilitando la MCDRAM en el programa con nivel de optimización *Opt-5*, el cual es mucho más demandante de ancho de banda, y como resultado, se obtuvo una diferencia de rendimiento de 5.31x. De esta forma, queda en evidencia que la MCDRAM no siempre mejorará el rendimiento en todos los casos, en realidad esto dependerá de distintos factores, siendo el principal la demanda de ancho de banda por parte del programa.

Al pasar de cómputo escalar a vectorizado en las versiones *Opt-2*, *Opt-3* y *Opt-4*, utilizando registros SIMD de 128 bits (SSE), 256 bits (AVX) y 512 bits (AVX-512) respectivamente, el rendimiento entregado se disparó de forma escalonada entre las tres versiones, con una mejora de rendimiento acumulada de 6.42x con respecto a la versión *Opt-1*.

La versión *Opt-5* trajo consigo una mejora de rendimiento de 1.14x por el solo hecho de reservar espacio en memoria para los vectores de forma alineada.

Con la versión *Opt-6* y su predicción de saltos por software se consiguió un fuerte incremento en el rendimiento, gracias a haber eliminado la combinación nociva que representaba la mala predicción de saltos con el *prefetching* de escritura. Con esta optimización, se logró evitar que continuamente se cargue en caché datos que rara vez son accedidos, evitando así dejar fuera a otros datos que sí integran el *working set* del programa. Con el uso de esta técnica de predicción de saltos por software, se logró la importante mejora de rendimiento de 2.68x.

La siguiente optimización aplicada al programa (*Opt-7*), fue la implementación de un desenrollado de bucles para dos de los *for* del algoritmo; uno con desenrollado manual (el bucle intermedio), y el otro con desenrollado por directivas (para el bucle más interno). La mejora de rendimiento obtenida fue de 1.36x.

Por último, la versión final con mayor grado de optimización, aplicó una afinidad de hilos con núcleos haciendo uso de la variable de entorno KMP_AFFINITY, esta última perteneciente a la interfaz de afinidad de hilos de Intel. La mejora de rendimiento en este caso fue de solo 1.005x.

En la tabla 4.12 se presentan sintéticamente los resultados de cada versión, con los parámetros BS y T que resultaron óptimos en cada caso.

Tabla 4.12: Tabla de resumen de versiones y sus resultados

Version	**Descripción de optimización**	**GFLOPS pico**	**GFLOPS promedio**	**Tasa de mejora**	**Tasa de m. acum.**	**BS ópt.**	**T ópt.**
Naive-Par	FW “clásico” paralel.	10	**10**	- - -	- - -	- - -	64
Opt-0	FW con *blocking* paralel.	31	**29**	2.99x	2.99x	64	256
Opt-1	Uso de la MCDRAM	31	**30**	1.02x	3.05x	64	256
Opt-2	Vect. con SSE	50	**47**	1.6x	4.88x	128	256
Opt-3	Vect. con AVX2	112	**103**	2.2x	10.74x	64	256
Opt-4	Vect. con AVX-512	230	**199**	1.96x	21.05x	128	128
Opt-5	Alineamiento de datos	365	**222**	1.14x	24.00x	128	64
Opt-6	Predicción de saltos	767	**600**	2.68x	64.32x	128	128
Opt-7	Desenrollado de bucles	1033	**821**	1.36x	87.48x	64 / 128	128
Opt-8	Afinidad de hilos	1039	**825**	1.005x	87.92x	64 / 128	128

Una vez obtenida la versión con mayor grado de optimización (*Opt-8*), se procedió a experimentar con variantes del algoritmo, utilizando de base a dicha versión (*Opt-8*). Primero se evaluó el rendimiento al omitir el cálculo de la matriz de reconstrucción de caminos mínimos (*Var-Nopath*), obteniendo con esta variante una mejora de 1.1x. Luego, fue probado el rendimiento*[1] para distintos grados de densidad del grafo de entrada (*Var-Dens*), oscilando los GFLOPS obtenidos entre 383 y 767 para el tamaño de entrada más grande (N=64K), pero siendo más estables con los tamaños de entrada más chicos. Se prosiguió probando con flags específicas al compilador con el fin de relajar la precisión en los cálculos sobre punto flotante (*Var-Prec*), y de esta forma, ganar rendimiento. No obstante, el *assembler* generado fue idéntico, por lo que dicha variante fue finalmente descartada. Por último, se evaluó el rendimiento obtenido para grafos con el tipo de dato *double* (Var-Double), con los cuales hubo una pérdida de rendimiento cercana al 50%.

1 Las pruebas en este caso se ejecutaron directamente con la implementación de la versión *Opt-8*.

Algo importante a remarcar, es que la combinación óptima de BS y cantidad de hilos (T) tuvo que ser re-evaluada en cada versión. Al aumentar o disminuir la demanda de ancho de banda de memoria entre versión y versión, hace que varían sensiblemente los parámetros óptimos. De hecho, para *Opt-0/1*, *Opt-2*, *Opt-3*. *Opt-4*, *Opt-5* y *Opt-6/7/8*, la combinación óptima de BS y T fue distinta en cada caso.

Capítulo 5

Conclusiones y trabajos futuros

Desde hace años, los aceleradores van tomando un protagonismo cada vez más relevante en la comunidad de HPC, siendo claves para lograr grandes niveles de rendimiento y de eficiencia energética. Con la introducción de los aceleradores Xeon Phi por parte de Intel, se provee a dicha comunidad de un nuevo tipo de acelerador, compatible con el conjunto de instrucciones x86. Esta característica atrae gran interés al permitir reutilizar técnicas y herramientas de programación preexistentes para procesadores x86, así como también brinda la posibilidad de correr directamente programas x86 también preexistentes, con pocas o ninguna modificación necesaria. En los últimos años, Intel ha lanzado la segunda generación de aceleradores Xeon Phi, con nombre en código KNL. Esta nueva generación trae importantes mejoras con respecto a su predecesora, tales como la implementación de la ejecución *fuera de orden*, la inclusión de una memoria de alto ancho de banda llamada MCDRAM, y a su vez mantiene la posibilidad de acceder directamente a la memoria DDR. Esta última característica es posible gracias a que se trata de un procesador *host* autónomo, el cual no tiene necesidad de ningún tipo de *off-loading* para su funcionamiento.

Una de las áreas bien conocidas por demandar gran poder de cómputo es la teoría de grafos, siendo un caso muy conocido el algoritmo FW para cómputo de caminos mínimos. Este algoritmo es ampliamente utilizado en diversos ámbitos, tales como el análisis del tráfico automovilístico, redes de computadoras, bioinformática, entre otros. Su popularidad y su alta demanda computacional lo vuelve un caso interesante de análisis en HPC. Es por este motivo, que en esta tesina se propuso como objetivo evaluar el uso de la arquitectura Xeon Phi KNL para acelerar el algoritmo FW.

A lo largo de este trabajo, se presentó al Xeon Phi KNL con su arquitectura, se enunció el problema del camino mínimo en grafos, y al algoritmo FW como uno de los algoritmos que lo resuelven. Con el fin de lograr la versión de FW más optimizada para el KNL se llevó a cabo un desarrollo incremental iterativo; es decir, partiendo de un algoritmo base, se aplicaron diferentes técnicas de optimización una a una, comparando los resultados de rendimiento en cada caso. En una segunda etapa, se procedió a desarrollar versiones variantes del programa, a modo de evaluar diferentes alternativas y parámetros. Entre ellos, se encuentra la variante que omite el cómputo de la matriz de reconstrucción de caminos mínimos, y el experimento utilizado para comparar el rendimiento entregado con grafos de entrada de distintos grados de densidad.

Luego de haber analizado los resultados de cada versión, de estas se pueden obtener las siguientes conclusiones del trabajo experimental:

- El uso de técnicas de *blocking* representa un pilar fundamental para lograr buen rendimiento.

- El uso de la memoria MCDRAM tiene un gran impacto en el rendimiento, aunque varía fuertemente de versión en versión según la demanda de ancho de banda que se tenga en cada una.
- La vectorización involucró fuertes ganancias de rendimiento. Siendo el KNL un procesador centrado en el procesamiento vectorial, resulta esencial lograr una correcta vectorización en el programa.
- El uso de técnicas de predicción de saltos por software sorprendentemente tuvo un impacto mayúsculo en el rendimiento. Resulta altamente recomendable estudiar la tasa de hits que tiene el algoritmo en los IF de su *hotspot*, para luego probar el rendimiento al predecir la condición del salto, tanto por el camino positivo como por el negativo.
- El desenrollado de bucles, si bien solo involucró una ganancia de rendimiento moderada, su relativa simplicidad de implementación lo vuelve una optimización atractiva para FW.
- En los casos en que no se requiera la matriz de reconstrucción de caminos mínimos, se puede ganar bastante rendimiento si en el algoritmo se omite el cómputo de la misma.
- El rendimiento entregado no varía demasiado en función del grado de densidad del grafo de entrada.
- Algunas de las optimizaciones evaluadas no impactaron en el rendimiento o, al menos, no fue significativo. Entre ellas, el alineamiento de datos en memoria, la afinidad de hilos y la relajación de la precisión de punto flotante. Aún así, afortunadamente son sencillas de implementar, y pueden ser consideradas a futuro, a pesar de su relativamente pequeño beneficio.
- Si se necesita trabajar a un nivel de precisión que haga necesario el uso del tipo de dato *double*, se debe pensar en que el rendimiento caerá aproximadamente a la mitad del entregado con *float*.
- Por último, los parámetros óptimos pueden variar fuertemente entre versiones luego de aplicar cada optimización. Siendo (en general) los mas influyentes para el rendimiento: el tamaño de bloque (BS) y la cantidad de hilos (T).

Habiendo logrado una versión de FW optimizada para Xeon Phi KNL, se considera que se ha cumplido con el objetivo planteado originalmente en esta tesina. Inicialmente, se investigó y documentó acerca de la arquitectura, el algoritmo, y las distintas técnicas de optimización que se fueron aplicando hasta lograr la versión óptima. La implementación desarrollada en este trabajo fue capaz de alcanzar un pico de rendimiento de 1039 GFLOPS, y se encuentra disponible en un repositorio web público para beneficio de la comunidad científica.

Como trabajos futuros, se pueden mencionar las siguientes ideas de investigación:

- Continuar refinando la optimización del programa desarrollado en este trabajo, ya que es probable que se le pueda extraer aún más rendimiento al KNL con este algoritmo. Por ejemplo, queda pendiente buscar nuevas optimizaciones y ajustes observando más detalladamente el *assembler* generado por el compilador.
- Siendo las GPU los aceleradores dominantes en la actualidad, resulta interesante utilizar el algoritmo FW para comparar el rendimiento entre una solución con GPU y la correspondiente con Xeon Phi desarrollada en este trabajo.

Referencias

[1] W. Feng, «The Importance of Being Low Power in High Performance Computing», *Los Alamos National Laboratory*, ago. 2005, [En línea]. Disponible en: https://icl.utk.edu/ctwatch/quarterly/articles/2005/08/the-importance-of-being-low-power-in-high-performance-computing/index.html.

[2] W. Feng, X. Feng, y R. Ge, «Green Supercomputing Comes of Age», *IT Professional*, vol. 10, n.º 1, pp. 17-23, feb. 2008.

[3] R. W. Floyd, «Algorithm 97: Shortest path», *Communications of the ACM*, p. 345, jun. 1962.

[4] S. Warshall, «A Theorem on Boolean Matrices», *Journal of the ACM*, pp. 11-12, jun. 1962.

[5] S. E. Jalali y M. Noroozi, «Determination of the optimal escape routes of underground mine networks in emergency cases», Safety Science, 2009, pp. 1077-1082.

[6] P. Khan, N. Chakraborty, y G. Konar, «Modification of Floyd-Warshall's algorithm for shortest path routing in wireless sensor networks», dic. 2014, pp. 1-6.

[7] A. Nakaya, S. Goto, y M. A. Kenehisa, «Extraction of correlated gene clusters by multiple graph omparison», *Genome Informatics*, pp. 44-53, 2001.

[8] J. Reinders, J. Jeffers, y A. Sodani, *Intel Xeon Phi Processor High Performance Programming*, Knights Landing Edition. Elsevier Inc, 2016.

[9] P. Bolzhauser, A. Sulistio, G. Angst, y C. Reich, «Parallelized Critical Path Search in Electrical Circuit Designs», 2009, [En línea]. Disponible en: https://www.academia.edu/4835295/Parallelized_Critical_Path_Search_in_Electrical_Circuit_Designs.

[10] M. Gong, G. Li, Z. Wang, L. Ma, y D. Tian, «An efficient shortest path approach for social networks based on community structure», 2016, [En línea]. Disponible en: https://www.sciencedirect.com/science/article/pii/S2468232216000123.

[11] M. Sniedovich, «Dynamic Programming and Principles of Optimality», *Journal of Mathematical Analysis and Applications*, n.º 65, pp. 566-606, 1978.

[12] G. Venkataraman, S. Sahni, y S. Mukhopadhyaya, «A Blocked All-Pairs Shortest-Paths Algorithm», *Journal of Experimental Algorithmics*, vol. 8, mar. 2002, doi: 10.1145/996546.996553.

[13] J.-S. Park, M. Penner, y V. K. Prasanna, «Optimizing graph algorithms for improved cache performance», *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, n.º 9, pp. 769-782, sep. 2004, doi: 10.1109/TPDS.2004.44.

[14] S. Han y S. Kang, *Optimizing All-Pairs Shortest-Path Algorithm Using Vector Instructions*. .

[15] S.-C. Han, F. Franchetti, y M. Püschel, «Program generation for the all-pairs shortest path problem», en *2006 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, sep. 2006, pp. 222-232.

[16] T. Srinivasan, R. Balakrishnan, S. A. Gangadharan, y V. Hayawardh, «A scalable parallelization of all-pairs shortest path algorithm for a high performance cluster environment», en *2007 International Conference on Parallel and Distributed Systems*, dic. 2007, pp. 1-8, doi: 10.1109/ICPADS.2007.4447721.

[17] M. Jian, L. Ke-ping, y L. Zhang, «A Parallel Floyd-Warshall algorithm based on TBB», en *2010 2nd IEEE International Conference on Information Management and Engineering*, abr. 2010, pp. 429-433, doi: 10.1109/ICIME.2010.5477752.

[18] E. Solomonik, A. Buluç, y J. Demmel, «Minimizing Communication in All-Pairs Shortest Paths», en *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, may 2013, pp. 548-559, doi: 10.1109/IPDPS.2013.111.

[19] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, y P. Sadayappan, «Parallel FPGA-based all-pairs shortest-paths in a directed graph», en *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, abr. 2006, p. 10 pp.-, doi: 10.1109/IPDPS.2006.1639347.

[20] Z. Chen, R. Pittman, y A. Forin, «Combining multicore and reconfigurable instruction set extensions», ene. 2010, pp. 33-36, doi: 10.1145/1723112.1723119.

[21] P. Harish y P. J. Narayanan, «Accelerating Large Graph Algorithms on the GPU Using CUDA», en *High Performance Computing – HiPC 2007*, Berlin, Heidelberg, 2007, pp. 197-208, doi: 10.1007/978-3-540-77220-0_21.

[22] G. J. Katz y J. T. Kider, «All-pairs shortest-paths for large graphs on the GPU», en *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, Sarajevo, Bosnia and Herzegovina, jun. 2008, pp. 47–55, Accedido: jun. 17, 2020. [En línea].

[23] B. Lund y J. W. Smith, «A Multi-Stage CUDA Kernel for Floyd-Warshall», *arXiv:1001.4108 [cs]*, feb. 2010, Accedido: jun. 18, 2020. [En línea]. Disponible en: http://arxiv.org/abs/1001.4108.

[24] K. Matsumoto, N. Nakasato, y S. G. Sedukhin, «Blocked All-Pairs Shortest Paths Algorithm for Hybrid CPU-GPU System», en *2011 IEEE International Conference on High Performance Computing and Communications*, sep. 2011, pp. 145-152, doi: 10.1109/HPCC.2011.28.

[25] H. Djidjev, G. Chapuis, R. Andonov, S. Thulasidasan, y D. Lavenier, «All-Pairs Shortest Path algorithms for planar graph for GPU-accelerated clusters», *Journal of Parallel and Distributed Computing*, vol. 85, pp. 91-103, nov. 2015, doi: 10.1016/j.jpdc.2015.06.008.

[26] K. Hou, H. Wang, y W. Feng, «Delivering Parallel Programmability to the Masses via the Intel MIC Ecosystem: A Case Study», en *2014 43rd International Conference on Parallel Processing Workshops*, sep. 2014, pp. 273-282, doi: 10.1109/ICPPW.2014.44.

[27] E. Rucci, A. De Giusti, y M. Naiouf, «Blocked All-Pairs Shortest Paths Algorithm on Intel Xeon Phi KNL Processor: A Case Study», La Plata - Buenos Aires - Argentina, 2017, pp. 47-57, [En línea]. Disponible en: http://sedici.unlp.edu.ar/handle/10915/63651.

[28] M. Klemm, «SIMD Vectorization with OpenMP». Intel, [En línea]. Disponible en: https://doc.itc.rwth-aachen.de/download/attachments/28344675/SIMD+Vectorization+with+OpenMP.PDF.